Randall Pruim
Nicholas J. Horton
Daniel T. Kaplan

Start Teaching with R

Project MOSAIC

Copyright (c) 2015 by Randall Pruim, Nicholas J. Horton, & Daniel T. Kaplan.

Edition 1.1, November 2015

This material is copyrighted by the authors under a Creative Commons Attribution 3.0 Unported License. You are free to *Share* (to copy, distribute and transmit the work) and to *Remix* (to adapt the work) if you attribute our work. More detailed information about the licensing is available at this web page: `http://www.mosaic-web.org/go/teachingRlicense.html`.

Cover Photo: Maya Hanna

Contents

About These Notes

We present an approach to teaching introductory and intermediate statistics courses that is tightly coupled with computing generally and with R and RStudio in particular. These activities and examples are intended to highlight a modern approach to statistical education that focuses on modeling, resampling based inference, and multivariate graphical techniques. A secondary goal is to facilitate computing with data through use of small simulation studies and appropriate statistical analysis workflow. This follows the philosophy outlined by Nolan and Temple Lang[1]. The importance of modern computation in statistics education is a principal component of the recently adopted American Statistical Association's curriculum guidelines[2].

Throughout this book (and its companion volumes), we introduce multiple activities, some appropriate for an introductory course, others suitable for higher levels, that demonstrate key concepts in statistics and modeling while also supporting the core material of more traditional courses.

A Work in Progress

These materials were originally developed for a workshop entitled *Teaching Statistics Using R* prior to the 2011 United States Conference on Teaching Statistics and revised for USCOTS 2011, USCOTS 2013, eCOTS 2014, ICOTS 9, and USCOTS 2015. We organized these workshops to help instructors integrate R (as well as some related technologies) into statistics courses at all levels. We received great feedback and many wonderful ideas from the participants and those that we've shared this with since the workshops.

Caution!
Despite our best efforts, you WILL find bugs both in this document and in our code. Please let us know when you encounter them so we can call in the exterminators.

[1] D. Nolan and D. Temple Lang. Computing in the statistics curriculum. *The American Statistician*, 64(2):97–107, 2010

[2] ASA Undergraduate Guidelines Workgroup. 2014 curriculum guidelines for undergraduate programs in statistical science. Technical report, American Statistical Association, November 2014. `http://www.amstat.org/education/curriculumguidelines.cfm`

We appreciate any feedback you are willing to share as we continue to work on these materials and the accompanying `mosaic` package. Drop us an email at `pis@mosaic-web.org` with any comments, suggestions, corrections, etc.

Updated versions will be posted at `http://mosaic-web.org`.

Two Audiences

We initially developed these materials for instructors of statistics at the college or university level. Another audience is the students these instructors teach. Some of the sections, examples, and exercises are written with one or the other of these audiences more clearly at the forefront. This means that

1. Some of the materials can be used essentially as is with students.

2. Some of the materials aim to equip instructors to develop their own expertise in R and RStudio to develop their own teaching materials.

Although the distinction can get blurry, and what works “as is" in one setting may not work “as is" in another, we’ll try to indicate which parts fit into each category as we go along.

R, RStudio and R Packages

R can be obtained from `http://cran.r-project.org/`. Download and installation are quite straightforward for Mac, PC, or linux machines.

RStudio is an integrated development environment (IDE) that facilitates use of R for both novice and expert users. We have adopted it as our standard teaching environment because it dramatically simplifies the use of R for instructors and for students. RStudio can be installed as a desktop (laptop) application or as a server application that is accessible to users via the Internet.

In addition to R and RStudio, we will make use of several packages that need to be installed and loaded separately. The `mosaic` package (and its dependencies) will

More Info

Several things we use that can be done only in RStudio, for instance `manipulate()` or RStudio’s integrated support for reproducible research).

RStudio server version works well with starting students. All they need is a web browser, avoiding any potential problems with oddities of students’ individual computers.

be used throughout. Other packages appear from time to time as well.

Marginal Notes

Marginal notes appear here and there. Sometimes these are side comments that we wanted to say, but we didn't want to interrupt the flow to mention them in the main text. Others provide teaching tips or caution about traps, pitfalls and gotchas.

Have a great suggestion for a marginal note? Pass it along.

What's Ours Is Yours – To a Point

This material is copyrighted by the authors under a Creative Commons Attribution 3.0 Unported License. You are free to *Share* (to copy, distribute and transmit the work) and to *Remix* (to adapt the work) if you attribute our work. More detailed information about the licensing is available at this web page: `http://www.mosaic-web.org/go/teachingRlicense.html`.

Document Creation

This document was created on November 30, 2015, using

- `knitr`, version 1.11
- `mosaic`, version 0.12.9003
- `mosaicData`, version 0.12.9003
- R version 3.2.2 Patched (2015-10-06 r69484)

Inevitably, each of these will be updated from time to time. If you find that things look different on your computer, make sure that your version of R and your packages are up to date and check for a newer version of this document.

DIGGING DEEPER

If you know LaTeX as well as R, then `knitr` provides a nice solution for mixing the two. We used this system to produce this book. We also use it for our own research and to introduce upper level students to reproducible analysis methods. For beginners, we introduce `knitr` with RMarkdown, which produces PDF, HTML, or Word files using a simpler syntax.

Project MOSAIC

This book is a product of Project MOSAIC, a community of educators working to develop new ways to introduce mathematics, statistics, computation, and modeling to students in colleges and universities.

The goal of the MOSAIC project is to help share ideas and resources to improve teaching, and to develop a curricular and assessment infrastructure to support the dissemination and evaluation of these approaches. Our goal is to provide a broader approach to quantitative studies that provides better support for work in science and technology. The project highlights and integrates diverse aspects of quantitative work that students in science, technology, and engineering will need in their professional lives, but which are today usually taught in isolation, if at all.

In particular, we focus on:

Modeling The ability to create, manipulate and investigate useful and informative mathematical representations of a real-world situations.

Statistics The analysis of variability that draws on our ability to quantify uncertainty and to draw logical inferences from observations and experiment.

Computation The capacity to think algorithmically, to manage data on large scales, to visualize and interact with models, and to automate tasks for efficiency, accuracy, and reproducibility.

Calculus The traditional mathematical entry point for college and university students and a subject that still has the potential to provide important insights to today's students.

Drawing on support from the US National Science Foundation (NSF DUE-0920350), Project MOSAIC supports a number of initiatives to help achieve these goals, including:

Faculty development and training opportunities, such as the USCOTS 2011, USCOTS 2013, eCOTS 2014, and ICOTS 9 workshops on *Teaching Statistics Using* R *and* RStudio, our 2010 Project MOSAIC kickoff workshop at the Institute for Mathematics and its Applications, and our *Modeling: Early and Often in Undergraduate Calculus* AMS PREP workshops offered in 2012, 2013, and 2015.

M-casts, a series of regularly scheduled webinars, delivered via the Internet, that provide a forum for instructors to share their insights and innovations and to develop collaborations to refine and develop them. Recordings of M-casts are available at the Project MOSAIC web site, `http://mosaic-web.org`.

The construction of syllabi and materials for courses that teach MOSAIC topics in a better integrated way. Such courses and materials might be wholly new constructions, or they might be incremental modifications of existing resources that draw on the connections between the MOSAIC topics.

More details can be found at `http://www.mosaic-web.org`. We welcome and encourage your participation in all of these initiatives.

Computational Statistics

There are at least two ways in which statistical software can be introduced into a statistics course. In the first approach, the course is taught essentially as it was before the introduction of statistical software, but using a computer to speed up some of the calculations and to prepare higher quality graphical displays. Perhaps the size of the data sets will also be increased. We will refer to this approach as **statistical computation** since the computer serves primarily as a computational tool to replace pencil-and-paper calculations and drawing plots manually.

In the second approach, more fundamental changes in the course result from the introduction of the computer. Some new topics are covered, some old topics are omitted. Some old topics are treated in very different ways, and perhaps at different points in the course. We will refer to this approach as **computational statistics** because the availability of computation is shaping how statistics is done and taught. Computational statistics is a key component of **data science**, defined as the ability to use data to answer questions and communicate those results.

In practice, most courses will incorporate elements of both statistical computation and computational statistics, but the relative proportions may differ dramatically from course to course. Where on the spectrum a course lies will be depend on many factors including the goals of the course, the availability of technology for student use, the perspective of the text book used, and the comfort-level of the instructor with both statistics and computation.

Students need to see aspects of computation and data science early and often to develop deeper skills. Establishing precursors in introductory courses help them get started.

Among the various statistical software packages available, R is becoming increasingly popular. The recent addition of RStudio has made R both more powerful and more accessible. Because R and RStudio are free, they have become widely used in research and industry. Training in R and RStudio is often seen as an important additional skill that a statistics course can develop. Furthermore, an increasing number of instructors are using R for their own statistical work, so it is natural for them to use it in their teaching as well. At the same time, the development of R and of RStudio (an optional interface and integrated development environment for R) are making it easier and easier to get started with R.

We developed the `mosaic` R package (available on CRAN) to make certain aspects of statistical computation and computational statistics simpler for beginners, without limiting their ability to use more advanced features of the language. The `mosaic` package includes a modelling approach that uses the same general syntax to calculate descriptive statistics, create graphics, and fit linear models.

Information about the `mosaic` package, including vignettes demonstrating features and supplementary materials (such as this book) can be found at `https://cran.r-project.org/web/packages/mosaic`.

1
Some Advice on Getting Started With R

Learning R is a gradual process, and getting off to a good start goes a long way toward ensuring success. In this chapter we discuss some strategies and tactics for getting started teaching statistics with R. In subsequent chapters we provide more details about the (relatively few) R commands that students need to know and some additional information about R that is useful for instructors to know. Along the way we present some of our favorite examples that highlight the use of R, including some that can be used very early in a course.

The mosaic package includes a vignette outlining a possible minimalist set of R commands for teaching an introductory course.

1.1 Strategies

Each instructor will choose to start his or her course differently, but we offer the following strategies (followed by some tactics and examples) that can serve as a guide for starting the course in a way that prepares students for success with R.

1. Start right away.

   Do something with R on day 1. Do something else on day 2. Have students do something by the end of week 1 at the latest.

2. Illustrate frequently.
   Have R running every class period and use it as needed throughout the course so students can see what R does. Preview topics by showing before asking students to do things.
3. Teach R as a language. (But don't overdo it.)

TEACHING TIP
RMarkdown provides a easy way to create handouts or slides for your students. See *R Markdown: Integrating a Reproducible Analysis Tool into Introductory Statistics* by B Baumer *et al* for more about integrating RMarkdown into your course. For those already familiar with LaTeX, there is also knitr/LaTeXintegration in RStudio.

There is a bit of syntax to learn – so teach it explicitly.

- Emphasize that capitalization (and spelling) matter.
- Explain carefully (and repeatedly) the syntax of functions.

  Fortunately, the syntax is very straightforward. It consists of a function name followed by an opening parenthesis, followed by a comma-separated list of arguments (which may be named), followed by a closing parenthesis.

  ```
  functionname ( name1=arg1, name2=arg2, ... )
  ```

  Get students to think about what a function does and what it needs to know to do its job. Generally, the function name indicates what the function does. The arguments provide the function with the necessary information to do the task at hand.
- Every object in R has a type (class). Ask frequently: *What type of thing is this?*

  Students need to understand the difference between a variable and a data frame and also that there are different kinds of variables (`factor` for categorical data and `numeric` for numerical data, for example). Instructors and more advanced students will want to know about `vector` and `list` objects.

Give more details in higher level courses.

Upper level students should learn more about user-defined functions and language control structures such as loops and conditionals. Students in introductory courses don't need to know as much about the language.

4. "Less volume, more creativity." [Mike McCarthy, head coach, Green Bay Packers]

   Use a few methods frequently and students will learn how to use them well, flexibly, even creatively. Focus on a small number of data types: numerical vectors, character strings, factors, and data frames. Choose functions that employ a similar framework and style to increase the ability of students to transfer knowledge from one situation to another.

5. Find a way to have computers available for tests.

NOTE

This is one of the primary motivations behind our `mosaic` package, which seeks to make more things simpler and more similar to each other so that students can more easily become independent, creative users of R. But even if you don't choose to do things exactly the way we do, we recommend using "Less Volume, More Creativity" as a guideline.

It makes the test match the rest of the course and is a great motivator for students to learn R. It also changes what you can ask for and about on tests.

One of us first did this at the request of students in an introductory statistics course who asked if there was a way to use computers during the test *"since that's how we do all the homework*." He now has students bring laptops to class for tests. Another of us has both in-class (without computer) and out-of-class (with computer) components to his assessment.

6. Rethink your course.

   If you have taught computer-free or computer-light courses in the past, you may need to rethink some things. With ubiquitous computing, some things disappear from your course:

   - Reading statistical tables.

     Does anyone still consult a table for values of sin, or log? All three of us have sworn off the use of tabulations of critical values of distributions (since none of us use them in our professional work, why would we teach this to students?)
   - "Computational formulas".

     Replace them with computation. Teach only the most intuitive formulas. Focus on how they lead to intuition and understanding, *not* computation.
   - (Almost all) hand calculations.

   NOTE

   One of the main uses of calculators on the AP Statistics exams is for the calculation of p-values and related quantiles.

   At the same time, other things become possible that were not before:

   - Large data sets
   - Beautiful plots
   - Simulations and methods based on randomization and resampling
   - Quick computations
   - Increased focus on concepts rather than calculations

   Get your students to think that using the computer is just part of how statistics is done, rather than an add-on.

7. Keep the message as simple as possible and keep the commands accordingly simple.

It is important not to get too complicated too quickly. Early on, we typically use default settings and focus on the main ideas. Later, we may introduce fancier options as students become comfortable with simpler things (and often demand more).

Particularly when doing graphics, beware of distracting students with the sometimes intricate details of beautifying for publication. If the default behavior is good enough, go with it.

8. Anticipate computationally challenged students, but be confident that you are leading them down the right path.

   Some students pick up R very easily. In every course there will be a few students who struggle. To help them, focus on diagnosing what they don't know and how to help them "get it".

   In our experience, the computer is often a fall guy for other things the student does not understand. Because the computer gives immediate feedback, it reveals these misunderstandings. For example, if students are confused about the distinctions among variables, statistics, and observational units, they will have a difficult time providing the correct information to a plotting function. The student may blame R, but that is not the primary source of the difficulty. If you can diagnose the true problem, you will improve their understanding of statistics and fix R difficulties simultaneously.

   Even students with a solid understanding of the statistical concepts will encounter R errors that they cannot eliminate. Tell students to copy and paste R code and error messages into email when they have trouble. When you reply, explain how the error message helped you diagnose their problem and help them generalize your solution to other situations. See Chapter 7 for some of the common error messages and what they might indicate.

Teaching Tip
When introducing R code to students, we emphasize the following questions: *What do you want* R *to do for you?* and *What information must you provide, if* R *is going to do that?* The first question generally determines the function that will be used. The second determines the inputs to that function.

Teaching Tip
Tell your students to copy and paste error messages into email rather than describe them vaguely. It's a big time saver for everyone

1.2 Tactics

Students must learn to see before they can see to learn.

1. Introduce Graphics Early.

   Introduce graphics very early, so that students see that they can get impressive output from simple commands. Try to break away from their prior expectation that there is a "steep learning curve."

   Accept the defaults – don't worry about the niceties (good labels, nice breaks on histograms, colors) too early. Let them become comfortable with the basic graphics commands and then play (make sure it feels like play!) with fancying things up.

Keep in mind that just because the graphs are easy to make on the computer doesn't mean your students understand how to read the graphs. Use examples that will help students develop good habits for visualizing data.

2. Introduce Sampling and Randomization Early.

   Since sampling drives much of the logic of statistics, introduce the idea of a random sample very early, and have students construct their own random samples. The phenomenon of a sampling distribution can be introduced in an intuitive way, setting it up as a topic for later discussion and analysis.

In keeping with this advice, most of the examples in this book fall in the area of exploratory data analysis. The organization is chosen to develop gradually anunderstanding of R. See the companion volume *A Student's Guide to R* for a tour of commands used in the primary sorts analyses used in the first two undergraduate statistics courses. This companion volume is organized by types of data analyses and presumes some familiarity with the R language.

2
Getting Started with RStudio

RStudio is an integrated development environment (IDE) for R that provides an alternative interface to R that has several advantages over other the default R interfaces:

- RStudio runs on Mac, PC, and Linux machines and provides a simplified interface that *looks and feels identical on all of them.*

  The default interfaces for R are quite different on the various platforms. This is a distractor for students and adds an extra layer of support responsibility for the instructor.

- RStudio can run in a web browser.

  In addition to stand-alone desktop versions, RStudio can be set up as a server application that is accessed via the internet. Installation is straightforward for anyone with experience administering a Linux system. Once set up at your institution, students can start using RStudio by simply opening a website from a browser and logging in. No additional installation or configuration is required.

  The web interface is nearly identical to the desktop version. As with other web services, users login to access their account. If students logout and login in again later, even on a different machine, their session is restored and they can resume their analysis right where they left off. With a little advanced set up, instructors can save the history of their classroom R use and students can load those history files into their own environment.

- RStudio provides support for reproducible research.

NOTE
Using RStudio in a browser is like Facebook for statistics. Each time the user returns, the previous session is restored and they can resume work where they left off. Users can login from any device with internet access.

CAUTION!
The desktop and server version of RStudio are so similar that if you run them both, you will have to pay careful attention to make sure you are working in the one you intend to be working in.

RStudio makes it easy to include text, statistical analysis (R code and R output), and graphical displays all in the same document. The RMarkdown system provides a simple markup language and renders the results in HTML. The knitr/LaTeX system allows users to combine R and LaTeX in the same document. The reward for learning this more complicated system is much finer control over the output format. Depending on the level of the course, students can use either of these for homework and projects.

We typically introduce students to RMarkdown very early, requiring students to use it for assignments and reports. Handouts, exams, and books like this one are produced using knitr/LaTeX, and it is relatively easy for interested students to migrate to knitr from RMarkdown if they are interested.

Note
To use Markdown or knitr/LaTeX requires that the knitr package be installed on your system. See Section 6.3 for instructions on installing packages.

- RStudio provides an integrated support for editing and executing R code and documents.
- RStudio provides some useful functionality via a graphical user interface.

  RStudio is not a GUI for R, but it does provide a GUI that simplifies things like installing and updating packages; monitoring, saving and loading environments; importing and exporting data; browsing and exporting graphics; and browsing files and documentation.
- RStudio provides access to the manipulate package.

  The manipulate package provides a way to create simple interactive graphical applications quickly and easily.

While one can certainly use R without using RStudio, RStudio makes a number of things easier and we highly recommend using RStudio. Furthermore, since RStudio is in active development, we fully expect more useful features in the future.

2.1 *Setting up R and RStudio*

R can be obtained from http://cran.r-project.org/. Download and installation are pretty straightforward for

Mac, PC, or Linux machines. RStudio is available from `http://www.rstudio.org/`. RStudio can be installed as a desktop (laptop) application or as a server application that is accessible to others via the Internet.

2.1.1 *RStudio in the cloud*

We primarily use an online version of RStudio. RStudio is a innovative and powerful interface to R that runs in a web browser or on your local machine. Running in the browser has the advantage that you don't have to install or configure anything. Just login and you are good to go. Futhermore, RStudio will "remember" what you were doing so that each time you login (even on a different machine) you can pick up right where you left off. This is "R in the cloud" and works a bit like GoogleDocs or Facebook for R.

Your system administrator will likely need to set up your own installation of RStudio for your institution, but we can attest that the process is straightforward and greatly facilitates student and faculty use.

2.1.2 RStudio *on your computer*

There is also a stand-alone version of the RStudio environment that you can install on your desktop or laptop machine. This can be downloaded from `http://www.rstudio.org/`. This assumes that you have a version of R installed on your computer (see below for instructions to download this from CRAN). Even if your students are primarily or exclusively using the server version of RStudio in a browser, instructors may like to have the security blanket of a version that does not require access to the internet. But be warned, the two version look so similar that you may occasionally find yourself working in one of them when you intend to be in the other.

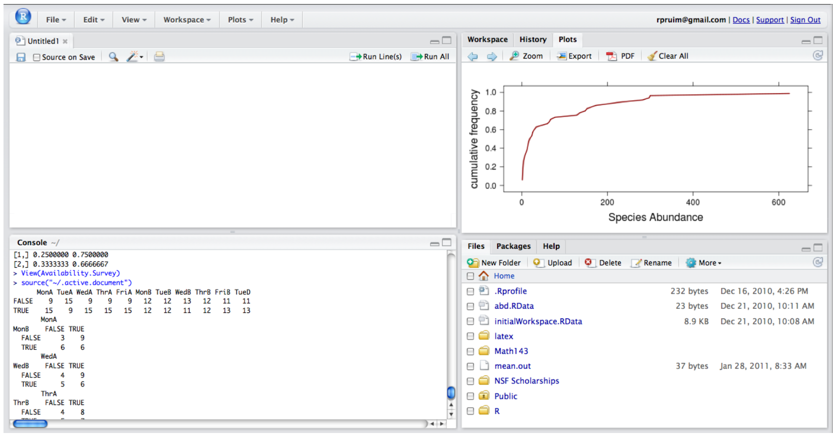
2.1.3 *Getting R from CRAN*

CRAN is the Comprehensive R Archive Network (`http://cran.r-project.org/`). You can download free versions of R for PC, Mac, and Linux from CRAN. (If you use the

RStudio stand-alone version, you also need to install R this way first.) All the instructions for downloading and installing are on CRAN. Just follow the appropriate instructions for your platform.

2.1.4 *Running RStudio the first time*

Once you have launched the desktop version of RStudio or logged in to an RStudio server, you will see something like the following.



```
[1,] 0.2500000 0.7500000
[2,] 0.3333333 0.6666667
> View(Availability.Survey)
> source("~/.active.document")
      MonA TueA WedA ThrA FriA MonB TueB WedB ThrB FriB TueD
FALSE    9   15    9    9    9   12   12   13   12   11   11
TRUE    15    9   15   15   15   12   12   11   12   13   13
       MonA
MonB    FALSE TRUE
  FALSE     3    9
  TRUE      6    6
       WedA
WedB    FALSE TRUE
  FALSE     4    9
  TRUE      5    6
       ThrA
ThrB    FALSE TRUE
  FALSE     4    8
```

.Rprofile	232 bytes	Dec 16, 2010, 4:26 PM
abd.RData	23 bytes	Dec 21, 2010, 10:11 AM
initialWorkspace.RData	8.9 KB	Dec 21, 2010, 10:08 AM
latex		
Math143		
mean.out	37 bytes	Jan 28, 2011, 8:33 AM
NSF Scholarships		
Public		
R		

Notice that RStudio divides its world into four panels. Several of the panels are further subdivided into multiple tabs. Which tabs appear in which panels can be customized by the user.

TEACHING TIP
We find it convenient to put the console in the upper left rather than the default location (lower left) so that students can see it better when we project our R session in class.

2.2 *Using R as a Calculator in the Console*

R can do much more than a simple calculator, and we will introduce additional features in due time. But performing simple calculations in R is a good way to begin learning the features of RStudio.

Commands entered in the Console tab are immediately executed by R. A good way to familiarize yourself with the console is to do some simple calculator-like computations. Most of this will work just like you would expect from a typical calculator. Try typing the following commands in the console panel.

```
5 + 3
```

```
[1] 8
```

```
15.3 * 23.4
```

```
[1] 358
```

```
sqrt(16)              # square root
```

```
[1] 4
```

This last example demonstrates how functions are called within R as well as the use of comments. Comments are prefaced with the # character. Comments can be very helpful when writing scripts with multiple commands or to annotate example code for your students.

You can save values to named variables for later reuse.

```
product = 15.3 * 23.4   # save result
product                 # display the result
```

```
[1] 358
```

```
product <- 15.3 * 23.4  # <- instead of =
product
```

```
[1] 358
```

TEACHING TIP
It's best to settle on using one or the other of the right-to-left assignment operators rather than to switch back and forth. Here we will adopt the arrow operator because it represents visually what is happening in an assignment, because it can also be used in a left to right manner, and because it makes a clear distinction between the assignment operator, the use of = to provide values to arguments of functions, and the use of == to test for equality.

Once variables are defined, they can be referenced in other operations and functions.

```
0.5 * product        # half of the product
```

```
[1] 179
```

```
log(product)         # (natural) log of the product
```

```
[1] 5.88
```

```
log10(product)       # base 10 log of the product
```

```
[1] 2.55
```

```
log2(product)        # base 2 log of the product
```

```
[1] 8.48
```

```
log(product, base=2) # another way for base 2 log
```

```
[1] 8.48
```

The semi-colon can be used to place multiple commands on one line. One frequent use of this is to save and print a value all in one go:

```
# store and show result
product <- 15.3 * 23.4; product
```

```
[1] 358
```

2.3 *Working with Files*

2.3.1 R *Script Files*

As an alternative, R commands can be stored in a file. RStudio provides an integrated editor for editing these files and facilitates executing some or all of the commands. To create a file, select File, then New File, then R Script from the RStudio menu. A file editor tab will open in the Source panel. R code can be entered here, and buttons and menu items are provided to run all the code (called sourcing the file) or to run the code on a single line or in a selected section of the file.

2.3.2 *RMarkdown, and knitr/LaTeX*

A third alternative is to take advantage of RStudio's support for reproducible research. If you already know LaTeX, you will want to investigate the `knitr`/LaTeX capabilities. For those who do not already know LaTeX, the simpler RMarkdown system provides an easy entry into the world of reproducible research methods. It also provides a good facility for students to create homework and reports that include text, R code, R output, and graphics.

To create a new RMarkdown file, select File, then New File, then RMarkdown. The file will be opened with a short template document that illustrates the mark up language. If you can click on From Template before creating the file, you will be given a list of template documents available in packages. If the the `mosaic` package is loaded, this list will include templates that make sure the mosaic package is loaded and change the defaults size for plots to be somewhat smaller than the generic RStudio default. The fancy version demonstrates many of the features of RMarkdown. (The RStudio web site includes extensive tutorials on using RMarkdown that demonstrate a wider range of features.) The plain templates are designed to quickly create new documents starting from a nearly blank slate.

The process of running the R code and combining text, R code, output, and graphics into a single file is called “knitting". Click on Knit to convert the RMarkdown document into an HTML, PDF, or Word file.

It is important to remember that unlike R scripts, which are executed in the console and have access to the console environment, RMarkdown and knitr/LaTeX files do not have access to the console environment. This is a good feature because it forces the files to be self-contained, which makes them transferable and respects good reproducible research practices. But beginners, especially if they adopt a strategy of trying things out in the console and copying and pasting successful code from the console to their file, will often create files that are incomplete and therefore do not compile correctly.

CAUTION!
RMarkdown, and knitr/LaTeX files do not have access to the console environment, so the code in them must be self-contained.

One good strategy for getting students to use RMarkdown is to provide them with an example document that includes the boiler plate you want them to use, loads any R packages that they will need, sets any knitr or R settings they way you prefer them, and has placeholders for the work you want them to do.

2.4 The Other Panels and Tabs

2.4.1 The History Tab

As commands are entered in the console, they appear in the History tab. These histories can be saved and loaded, there is a search feature to locate previous commands, and individual lines or sections can be transfered back to the console. Keeping the History tab open will allow students to look back and see the previous several commands. This can be especially useful when commands produce a fair amount of output and so scroll off the screen rapidly. History files can be saved and distributed to students so that they can rerun the code illustrated in class. (Before saving the history, you can remove any lines that you don't want saved to spare your students repeating all of your typing errors.)

An alternative is to produce RMarkdown files in class and make those available. This provides a better mechanism for adding additional comments or instructions.

2.4.2 Communication between tabs

RStudio provides several ways to move R code between tabs. Pressing the Run button in the editing panel for an R script or RMarkdown or other file will copy lines of code into the Console and run them.

2.4.3 The Files Tab

The Files tab provides a simple file manager. It can be navigated in familiar ways and used to open, move, rename, and delete files. In the browser version of RStudio, the Files tab also provides a file upload utility for moving files from the local machine to the server. In RMarkdown and knitr files one can also run the code in a particular chunk or in all of the chunks in a file. Each of these features makes it easy to try out code "live" while creating a document that keeps a record of the code.

In the reverse direction, code from the history can be copied either back into the console to run them again (perhaps after editing) or into one of the file editing tabs for inclusion in a file.

2.4.4 The Help Tab

The Help tab is where RStudio displays R help files. These can be searched and navigated in the Help tab. You can also open a help file using the ? operator in the console. For example

```
?log
```

Will provide the help file for the logarithm function.

2.4.5 The Environment Tab

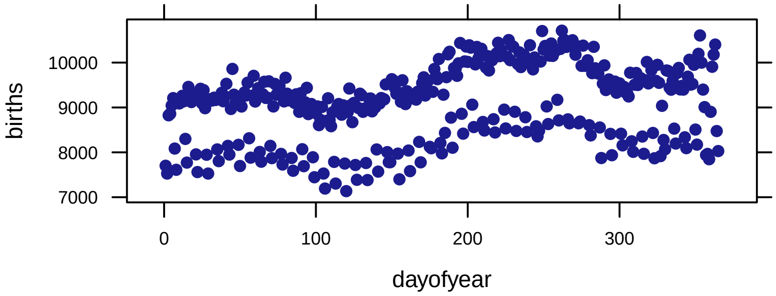
The Environment tab shows the objects available to the console. These are subdivided into data, values (non-data frame, non-function objects) and functions. The broom icon can be used to remove all objects from the environment, and it is good to do this from time to time, especially when running in RStudio server or if you choose to save the environment when shutting down RStudio since in these cases objects can stay in the environment essentially indefinitely.

2.4.6 The Plots Tab

Plots created in the console are displayed in the Plots tab. For example,

If you haven't been entering these example commands at your console, go back and do it!

```
# this will make lattice graphics available
require(mosaic)
xyplot( births ~ dayofyear, data=Births78)
```



will display the number of births in the United States for each day in 1978. From the Plots tab, you can navigate to previous plots and also export plots in various formats or copy them to the cliboard after interactively resizing them.

2.4.7 The Packages Tab

Much of the functionality of R is located in packages, many of which can be obtained from a central clearing house called CRAN (Comprehensive R Archive Network).

The Packages tab facilitates installing and loading packages. It will also allow you to search for packages that have been updated since you installed them.

3
Using R Early in the Course

This chapter includes some of our favorite activities for early in the course. These activities simultaneously provide the students with a first glimpse of R and an introduction to some major themes of the course. Used this way, it is not necessary for students to understand the details of the R code. Instead have them focus on the questions being asked on how the results presented shed light on the answers to these questions.

3.1 Coins and Cups: The Lady Tasting Tea

This section is a slightly modified version of a handout one of the authors has given Intro Stats students on Day 1 after going through the activity as a class discussion.

There is a famous story about a lady who claimed that tea with milk tasted different depending on whether the milk was added to the tea or the tea added to the milk. The story is famous because of the setting in which she made this claim. She was attending a party in Cambridge, England, in the 1920s. Also in attendance were a number of university dons and their wives. The scientists in attendance scoffed at the woman and her claim. What, after all, could be the difference?

All the scientists but one, that is. Rather than simply dismiss the woman's claim, he proposed that they decide how one should *test* the claim. The tenor of the conversation changed at this suggestion, and the scientists began to discuss how the claim should be tested. Within a few minutes cups of tea with milk had been prepared and presented to the woman for tasting.

At this point, you may be wondering who the innovative scientist was and what the results of the experiment were. The scientist was R. A. Fisher, who first described this situation as a pedagogical example in his 1925 book on statistical methodology [1]. Fisher developed statistical methods that are among the most important and widely used methods to this day, and most of his applications were biological.

You might also be curious about how the experiment came out. How many cups of tea were prepared? How many did the woman correctly identify? What was the conclusion?

Fisher never says. In his book he is interested in the method, not the particular results. But we can use this setting to introduce some key ideas in statistics.

Let's suppose we decide to test the lady with ten cups of tea. We'll flip a coin to decide which way to prepare the cups. If we flip a head, we will pour the milk in first; if tails, we put the tea in first. Then we present the ten cups to the lady and have her state which ones she thinks were prepared each way.

It is easy to give her a score (9 out of 10, or 7 out of 10, or whatever it happens to be). It is trickier to figure out what to do with her score. Even if she is just guessing and has no idea, she could get lucky and get quite a few correct – maybe even all 10. But how likely is that?

Teaching Tip
The score is setting up the idea of a test statistic for later, but there is no need to introduce that terminology on day 1.

Let's try an experiment. I'll flip 10 coins. You guess which are heads and which are tails, and we'll see how you do.

Teaching Tip
Have each student make a guess by writing down a sequence of 10 H's or T's while you flip the coin behind a barrier so that the students cannot see the results.

Comparing with your classmates, we will undoubtedly see that some of you did better and others worse.

Now let's suppose the lady gets 9 out of 10 correct. That's not perfect, but it is better than we would expect for someone who was just guessing. On the other hand, it is not impossible to get 9 out of 10 just by guessing. So here is Fisher's great idea: Let's figure out how hard it is to get 9 out of 10 by guessing. If it's not so hard to do, then perhaps that's just what happened, so we won't be too impressed with the lady's tea tasting ability. On the other hand, if it is really unusual to get 9 out of 10 correct by guessing, then we will have some evidence that she must be able to tell something.

But how do we figure out how unusual it is to get 9 out of 10 just by guessing? We'll learn another method

[1] R. A. Fisher. *Statistical Methods for Research Workers*. Oliver & Boyd, 1925

later, but for now, let's just flip a bunch of coins and keep track. If the lady is just guessing, she might as well be flipping a coin.

So here's the plan. We'll flip 10 coins. We'll call the heads correct guesses and the tails incorrect guesses. Then we'll flip 10 more coins, and 10 more, and 10 more, and . . . . That would get pretty tedious. Fortunately, computers are good at tedious things, so we'll let the computer do the flipping for us.

The `rflip()` function can flip one coin

```
require(mosaic)
rflip()
```

```
Flipping 1 coin [ Prob(Heads) = 0.5 ] ...

T

Number of Heads: 0 [Proportion Heads: 0]
```

Note

There is a subtle switch here. Before we were asking how many of the students H's and T's matched the flipped coin. Now we are using H to simulate a correct guess and T to simulate an incorrect guess. This makes simulating easier.

or a number of coins

```
rflip(10)
```

```
Flipping 10 coins [ Prob(Heads) = 0.5 ] ...

H T H H T H H H T H

Number of Heads: 7 [Proportion Heads: 0.7]
```

Typing `rflip(10)` a bunch of times is almost as tedious as flipping all those coins. But it is not too hard to tell R to `do()` this a bunch of times.

Note

Notice that `do()` is clever about what information it records. Rather than recording all of the individual tosses, it is only recording the number of flips, the number of heads, and the number of tails.

```
do(3) * rflip(10)
```

```
   n heads tails prop
1 10     8     2  0.8
2 10     4     6  0.4
3 10     1     9  0.1
```

Now let's get R to do() it for us 10,000 times and make a table of the results.

```
# store the results of 10000 simulated ladies
random.ladies <- do(10000) * rflip(10)
```

```
tally(~heads, data=random.ladies)
```

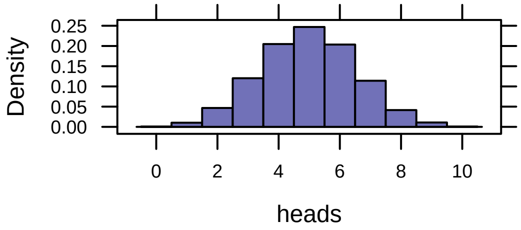
```
   0    1    2    3    4    5    6    7    8    9   10 
   5  102  467 1203 2048 2470 2035 1140  415  108    7 
```

```
# We can also display a table using percentages
tally(~heads, data=random.ladies, format="prop")
```

```
     0      1      2      3      4      5      6      7 
0.0005 0.0102 0.0467 0.1203 0.2048 0.2470 0.2035 0.1140 
     8      9     10 
0.0415 0.0108 0.0007 
```

We can display this table graphically using a plot called a **histogram** with bins of width 1.

```
histogram(~ heads, data=random.ladies, width=1)
```



You might be surprised to see that the number of correct guesses is exactly 5 (half of the 10 tries) only 25% of the time. But most of the results are quite close to 5 correct. For example, 67% of the results are 4, 5, or 6, for example. About 90% of the results are between 3 and 7 (inclusive). But getting 8 correct is a bit unusual, and getting 9 or 10 correct is even more unusual.

Teaching Tip

There is always the question of how many simulations to perform. This is a trade-off between speed and accuracy. For simple things, one can easily perform 10,000 or more simulations live in class. For more complicated things (that might require fitting a model and extracting information from it at each iteration) you might prefer a smaller number for live demonstrations.

When you cover inference for a proportion, it is a good idea to use those methods to revisit the question of how many replications are required in that context.

Note

The mosaic package adds some additional features to histogram(). In particular, the width and center arguments, which make it easier to control the bins, are only available if you are using the mosaic package.

So what do we conclude? It is possible that the lady could get 9 or 10 correct just by guessing, but it is not very likely (it only happened in about 1.2% of our simulations). So *one of two things must be true*:

- The lady got unusually “lucky", or
- The lady is not just guessing.

Although Fisher did not say how the experiment came out, others have reported that the lady correctly identified all 10 cups! [2]

A DIFFERENT DESIGN

Suppose instead that we prepare five cups each way (and that the woman tasting knows this). We give her five cards labeled “milk first”, and she must place them next to the cups that had the milked poured first. How does this design change things?

We could simulate this by shuffling a deck of 10 cards and dealing five of them.

```
cards <-
  factor(c("M","M","M","M","M","T","T","T","T","T"))
tally(~deal(cards, 5))
```

```
M T
3 2
```

NOTE
The use of `factor()` here lets R know that the possible values are ‘M’ and ‘T’, even when only one or the other appears in a given random sample.

```
results <- do(10000) * tally(~deal(cards, 5))
tally(~ M, data=results)
```

```
   0    1    2    3    4    5
  44  993 3966 3927 1028   42
```

```
tally(~ M, data=results, format="prop")
```

```
     0      1      2      3      4      5
0.0044 0.0993 0.3966 0.3927 0.1028 0.0042
```

```
tally(~ M, data=results, format="perc")
```

```
    0     1     2     3     4     5
 0.44  9.93 39.66 39.27 10.28  0.42
```

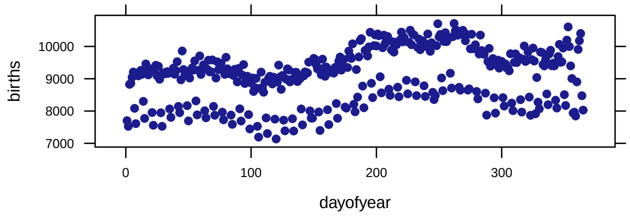
[2] D. Salsburg. *The Lady Tasting Tea: How statistics revolutionized science in the twentieth century*. W.H. Freeman, New York, 2001

3.2 Births by Day

The Births78 data set contains the number of births in the United States for each day of 1978. A scatter plot of births by day of year reveals some interesting patterns. Let's see how the number of births depends on the day of the year.

Note
The use of the phrase "depends on" is intentional. Later we will emphasize how y x can often be interpreted as "y depends on x".

```
xyplot(births ~ dayofyear, data=Births78)
```



Teaching Tip
The plot could also be made using date. For general purposes, this is probably the better plot to make, but using dayofyear forces students to think more about what the x-axis means.

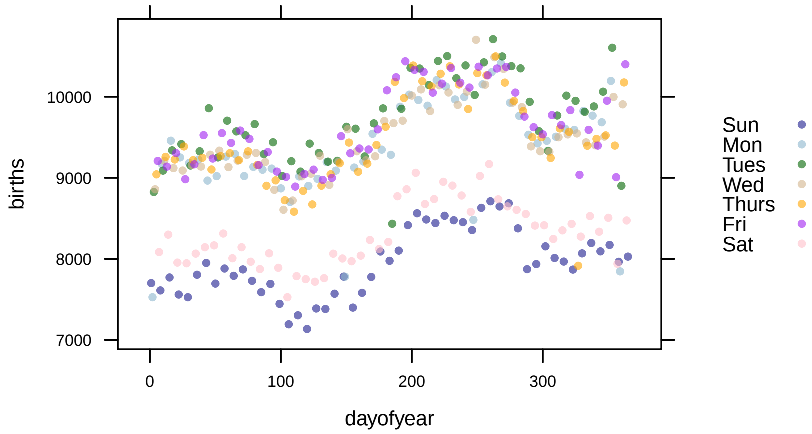
When shown this image, students should readily be able to describe two patterns in the data; they should notice both the rise and fall over the course of the year and the two "parallel waves". Many students will be able to come up with conjectures about the peaks and valleys, but they often struggle to correctly interpret the parallel waves. Having them make conjectures about this will quickly reveal whether they are correctly interpreting the plot.

Teaching Tip
This can make a good "think-pair-share" activity. Have students come up with possible explanations, then discuss these explanations with a partner. Finally, have some of the pairs share their explanations with the entire class.

One conjecture about the parallel waves can be checked using the data at hand. If we display each day of the week with a different symbol or color, we see that there are fewer births on weekends – likely because scheduled births are less likely on weekends. There are a handful of exceptions which are readily seen to be holidays.

```
require(mosaicData)       # load mosaic data sets
xyplot(births ~ dayofyear, data=Births78,
       groups=wday,
       auto.key=list(space="right"))
```

Teaching Tip
The handful of exceptions are easier to see if we "connect the dots". See Section 4.6.1.



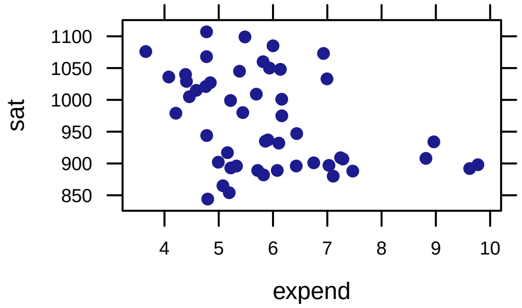
A discussion of this or some other data set that can be explored through graphical displays is a good way to demonstrate "statistical curiosity", to illustrate the power of R for creating graphs, and to introduce the importance of covariates in statistical analysis.

Visualization has been called the "gateway drug" to statistics. It can be a great way to lure students into statistics – and away from their graphing calculators.

3.3 SAT and Confounding

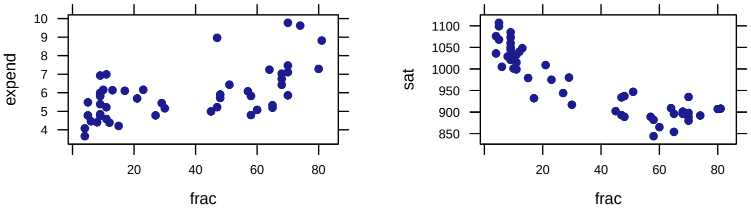
The SAT data set contains information about the link between SAT scores and measures of educational expenditures. Students are often surprised to see that states that spend more on education do worse on the SAT.

```
xyplot(sat ~ expend, data=SAT)
```



The implication, that spending less might give better results, is not justified. Expenditures are confounded with the proportion of students who take the exam, and scores are higher in states where fewer students take the exam.

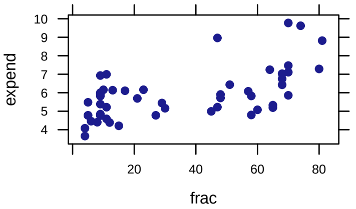
```
xyplot(expend ~ frac, data=SAT)
xyplot(sat ~ frac, data=SAT)
```

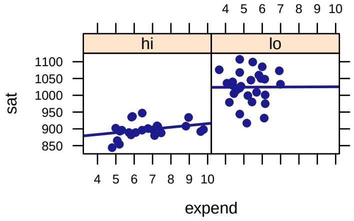


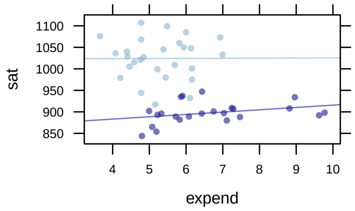
It is interesting to look at the original plot if we place the states into two groups depending on whether more or fewer than 40% of students take the SAT:

```
SAT <- mutate(SAT,
        fracGroup = derivedFactor(
          hi = (frac > 40),
          lo = (frac <=40) ))
```

```
xyplot(expend ~ frac, data=SAT)
xyplot( sat ~ expend | fracGroup , data=SAT,
        type=c("p","r") )
xyplot( sat ~ expend, groups = fracGroup , data=SAT,
        type=c("p","r") )
```







This example can be used to warn against interpreting relationships causally and to illustrate the importance of considering covariates.

3.4 *Mites and Wilt Disease*

This example shows how to build up to statistical inference from first principles.

Researchers suspect that attack of a plant by one organism induces resistance to subsequent attack by a different organism. Individually potted cotton plants were randomly allocated to two groups: infestation by spider mites or no infestation. After two weeks the mites were dutifully removed by a conscientious research assistant, and both groups were inoculated with Verticillium, a fungus that causes Wilt disease. The researchers were hoping the data would shed light on the following big question:

> Is there a relationship between infestation and Wilt disease?

The accompanying table shows a cross tabulation the number of plants that developed symptoms of Wilt disease.

```
tally(outcome ~ treatment, data = Mites, margins = TRUE)
```

```
         treatment
outcome   mites no mites
  no wilt    15        4
  wilt       11       17
  Total      26       21
```

Some questions for students:

1. What do you think is the explanatory variable? Response variable?

2. What proportion of the plants in the study with mites developed Wilt disease?

3. What proportion of the plants in the study with no mites developed Wilt disease?

4. Relative risk is the ratio of two risk proportions. What is the relative risk of developing Wilt disease, comparing mites to no mites?

5. If there were no association between mites and Wilt disease, what would the relative risk be (in the population as a whole)? How close is the relative risk computed from the data to this value?

6. Let X be the number of plants in the no mites group that did not develop Wilt disease. What are the possible values for X?

7. Assuming a population relative risk of 1, give two possible values for X that would be more unusual than the value for these data?

Questions 6-7 can be addressed using cards:

Physical Simulation

1. Select 47 cards from your deck: 26 red (mites!) and 21 black
2. Shuffle the cards well
3. Deal out 19 cards, these represent the 19 plants without Wilt disease.
4. Count the number of black cards among those 19. What do these represent?
5. Repeat steps 2 –4, five times.

Students can pool their results by recording them in a table on the board at the front of the room. Then have students process the results by answering the following questions.

8. How many black cards would we expect (on average)? Why?

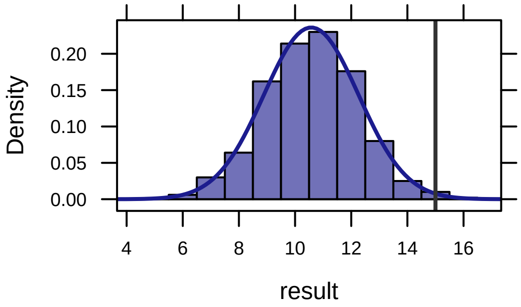
9. What did we observe?

10. How would we summarize these results? What is the big idea?

Once the simulation with cards has been completed, we can use R to do many more simulations very quickly.

Computational Simulation

```
tally(outcome ~ treatment, data=Mites)

         treatment
outcome   mites no mites
  no wilt    15        4
  wilt       11       17

X <- tally(outcome ~ treatment, data=Mites)[1,1]; X

[1] 15

nullDist <- do(1000) *
  tally(outcome ~ shuffle(treatment), data=Mites)[1,1]
histogram(~ result, data=nullDist, width=1,
          type="density", fit="normal", v=15)
```



4
Less Volume, More Creativity

> A lot of times you end up putting in a lot more volume, because you are teaching fundamentals and you are teaching concepts that you need to put in, but you may not necessarily use because they are building blocks for other concepts and variations that will come off of that ... In the offseason you have a chance to take a step back and tailor it more specifically towards your team and towards your players.
>
> – Mike McCarthy, Head Coach, Green Bay Packers

> Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.
>
> – Antoine de Saint-Exupery, writer, poet, aviator

One key to successfully introducing R is finding a set of commands that is

- small,
- coherent, and
- powerful.

This chapter provides an extensive example of this “Less Volume, More Creativity" approach. The `mosaic` package (combined with the `lattice` package and other core R functionality) provides a simple yet powerful framework that equips students to produce all of the

- numerical summaries,
- graphical summaries, and
- linear models

Mike McCarthy, head coach of the Green Bay Packers football team uses “Less Volume, More Creativity" as a mantra for his coaching staff as they prepare the game plan each week. As an illustration of the principle at work, when asked by a fan how many pass plays the team prepares for a given opponent, the coach answered, “When I first got into the NFL we had 150 passes in our game plan. I’ve put a sign on all of the coordinators’ doors – Less volume, more creativity. We function with more concepts with less volume. [Now] We’re more around 50 [passes] in a game plan.

needed in an introductory course. By presenting this as one master template with variations, we emphasize the similarity among these commands and reduce the cognitive load for students. In our experience, this has made R much more approachable and enjoyable for students and their instructors.

4.1 The `mosaic` package and the formula template

Much of the early work on the `mosaic` package centered on producing a minimal set of R commands that could provide students with everything need for introductory statistics without overwhelming students with too many commands. One of the `mosaic` package vignettes includes a document describing just such a set of commands.

Much of this is built off the following template that is used repeatedly

$$\boxed{}\left(\boxed{}\sim\boxed{}\text{, data = }\boxed{}\right)$$

The template is used by filling in the boxes. It helps to give each box a name:

$$\boxed{\text{goal}}\left(\boxed{\text{y}}\sim\boxed{\text{x}}\text{, data = }\boxed{\text{mydata}}\right)$$

The template has a bit more flexibility than we have indicated. Sometimes the `y` is not needed:

```
goal( ~ x, data=mydata )
```

Teaching Tip
After introducing this template, you might quiz students to make sure they have learned it. This will also emphasize its importance.

The formula may also include a third part

```
goal( y ~ x | z , data=mydata )
```

We can unify all of these into one form:

```
goal( formula , data=mydata )
```

The template can be applied to create numerical summaries, graphical summaries, or model fits by answering two questions and using the answers to fill in the slots of the template:

1. What do you want R to do?

   This is the goal.

2. What must R know to do that?

   These are the inputs to the function. For numerical summaries, graphical summaries, and model fits, we typically need to specify the variables involved and the data frame in which they are stored.

4.2 Graphical summaries of data

Teaching Tip
We recommend showing some plots on the first day and having students generate their own graphs before the end of the first week.

Graphical summaries are an important and eye-catching way to demonstrate the power and flexibility of our template. We like to introduce students to graphical summaries early in the course. This gives the students access to functionality where R really shines (and is certainly much better than a hand-held calculator). It also begins to develop their ability to interpret graphical representations of data, to think about distributions, and to pose statistical questions.

There are several ways to make graphs in R. One approach is a system called `lattice` graphics. Whenever the `mosaic` package is loaded, the `lattice` package is also loaded. One of the attractive aspects of `lattice` plots is that they make use of the same template we will use for numerical summaries and linear models.

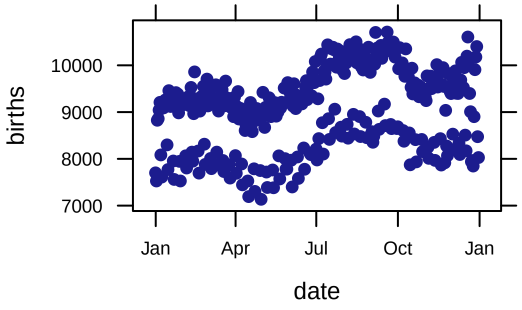
More Info
We are often asked about the other graphics systems, especially `ggplot2` graphics. In our experience, `lattice` makes it easier for beginners to create a wide variety of more or less "standard" plots – including the ability to represent multiple variables at once. `ggplot2`, on the other hand, makes it easier to generate custom plots or to combine plot components. Each has its place, and we use both systems. But for beginners, we typically emphasize `lattice`.

The new `ggvis` package, by the same author as `ggplot2` adds interactivity and speed to the strengths of `ggplot2`.

4.2.1 Graphical summaries of two variables

A first example: Making a scatter plot

As an example, let's create the following plot, which shows the number of births in the United States for each day in 1978.



TEACHING TIP
This plot can make an interesting discussion starter early in a course. Ask students to conjecture explanations for the patterns they observe in the plots. Their answers will reveal whether they are interpreting the plot correctly.

1. What is the goal?

   We want a scatter plot. The function that creates scatter plots is called `xyplot()`, so this will go into the goal slot of our template.

2. What does R need to know?

   R needs to know which variable goes where and where to find the variables. In this case, the data are stored in the `Births78` data frame:

```
head(Births78)
```

```
        date births dayofyear  wday
1 1978-01-01   7701         1   Sun
2 1978-01-02   7527         2   Mon
3 1978-01-03   8825         3  Tues
4 1978-01-04   8859         4   Wed
5 1978-01-05   9043         5 Thurs
6 1978-01-06   9208         6   Fri
```

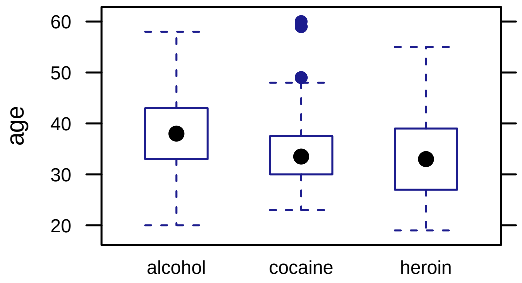
   We want to put the number of births (`births`) along the y-axis and the day of the year (`date`) along the x-axis.

Putting this all together, we get the following command

```
xyplot(births ~ date, data=Births78)
```

Another Example: Boxplots

Now let's create this plot, which shows boxplots of age for each of three substances abused by participants in the *Health Evaluation and Linkage to Primary Care* randomized clinical trial.



More Info

You can find out more about the HELPrct data set using the help command: ?HELPrct. This will provide you with the codebook for the data and links to the original source.

There are also a number of functions that allow us to inspect the contents of a data frame. Among our favorites are inspect(), glimpse(), and head().

The data we need are in the HELPrct data frame, from which we want to display variables age and substance on the *y*- and *x*-axes. According to our template, the command to create this plot has the form

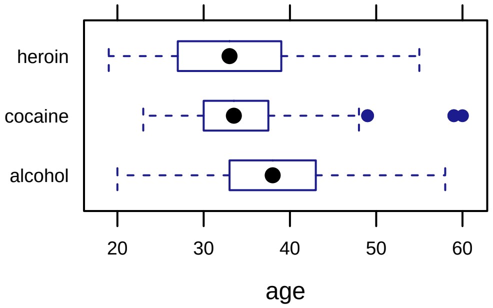
```
goal(age ~ substance, data=HELPrct)
```

The only additional information we need is the name of the function that creates boxplots. That function is bwplot(). So we can create the plot with

```
bwplot(age ~ substance, data=HELPrct)
```

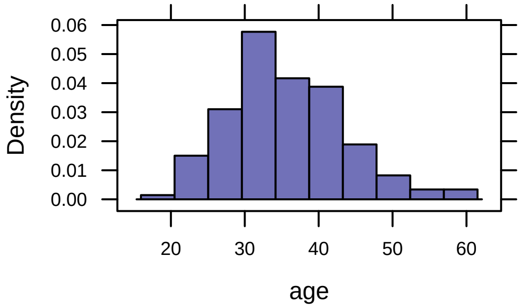
To make the boxplots horizontal instead of vertical, reverse the roles of age and substance:

```
bwplot(substance ~ age, data=HELPrct)
```



4.2.2 Graphical summaries of one variable

If we want to make a plot that involves only one variable, we simply omit the y-part of the formula. For example, a histogram like

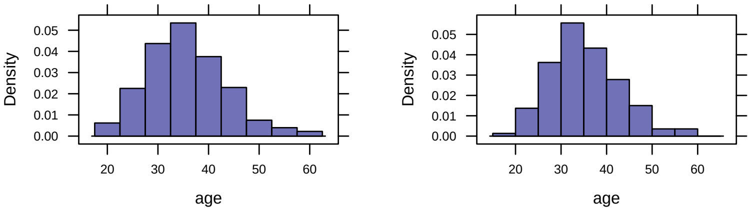


can be made with

```
histogram( ~ age, data=HELPrct)
```

The mosaic package adds some extra functionality to histogram() to make it easier to specify the bins used. In particular, the options width and center (default is 0) can be used to define the width of the bins and the center of one of the bins. For example, to create a histogram with bins that are 5 years wide we can use width=5, and we can shift the bins left and right by providing a value for center.

```
histogram( ~ age, data=HELPrct, width=5)
histogram( ~ age, data=HELPrct, width=5, center=2.5)
```



There is enough data here to use a bin for each integer if we like. Because the default value of center is 0, setting

More Info
You may be wondering about plots for two categorical variables. A commonly used plot for this is a segmented bar graph. We will treat this as a augmented version of a simple bar graph, which is a graphical summary of one categorical variable.

Another plot that can be used to display two (or more) categorical variables is a mosaic plot. The lattice package does not include mosaic plots, but the vcd package provides a mosaic() function that creates mosaic plots.

Caution!
It is important to note that when there is only one variable it is on the *right* side of the formula.

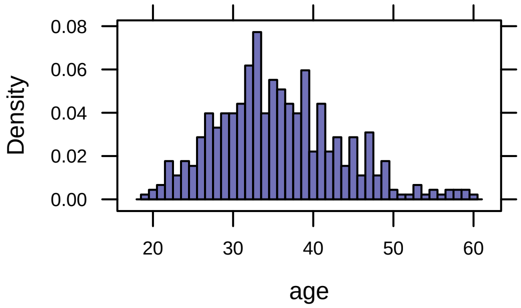
Teaching Tip
Tell students that because R is computing the y values, we don't need to provide them. This isn't exactly the reason why things are this way, but it helps them remember.

Introducing width and center here is perhaps a violation of our usual policy of accepting defaults and saving options for later. But it is important that histogram bins be chosen appropriately, and no algorithmic default works well for all data sets. We encourage students to make several histograms and to experiment with center and especially width.

Note
center need not be contained in the bins that are displayed. So to get bins with edges "on the 0's and 5's", we can set the center to 2.5, regardless of the range of the data.

width to 1 centers the bins on the integers, avoiding potential confusion about which edge is included in the bin.

```
histogram( ~ age, data=HELPrct, width=1)
```

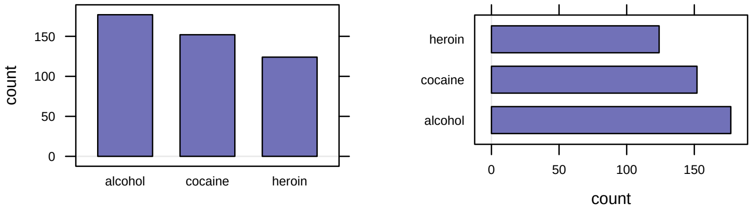


Additional plots of a single quantitative variable are illustrated in Section sec:paletteOfPlots.

For a single categorical variable, we can make a bar graph for a categorical variable using bargraph() in place of histogram(). Since formulas are required to have a right-hand side, horizontal bar graphs are produced using horizontal = TRUE.

More Info
The bargraph() function is not in the lattice package but in the mosaic package. The lattice function barchart() creates bar graphs from *summarized* data; bargraph() takes care of creating this summary data and then uses barchart() to create the plot.

```
bargraph( ~ substance, data=HELPrct)
bargraph( ~ substance, data=HELPrct, horizontal=TRUE)
```



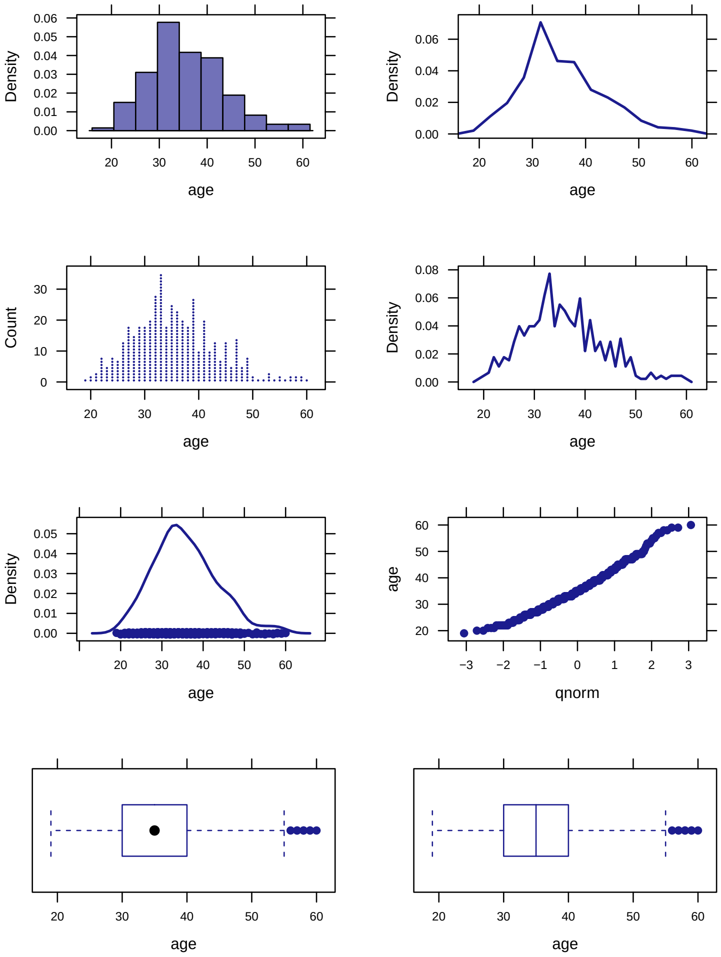
4.2.3 *A palette of plots*

The power of the template is that we can now make many different kinds of plots by mimicking the examples above but replacing the goal.

More Info
If you are unfamiliar with some of the plots, like ashplots and frequency polygons, keep reading. We have more to say about them shortly.

```
  histogram( ~ age, data=HELPrct)
freqpolygon( ~ age, data=HELPrct)
```

```
    dotPlot( ~ age, data=HELPrct, width=1)
    ashplot( ~ age, data=HELPrct, width=1)
densityplot( ~ age, data=HELPrct)
     qqmath( ~ age, data=HELPrct)
     bwplot( ~ age, data=HELPrct)
     bwplot( ~ age, data=HELPrct, pch = "|")
```



Note

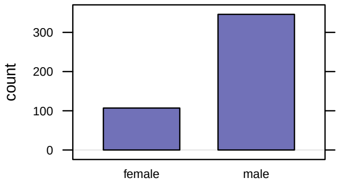
If you prefer the more traditional boxplot display with a line at the median rather than a dot, you can make that the default behavior with `trellis.par.set(box.dot = list(pch = "|"))`.

Some people prefer the more traditional boxplot display with a line at the median rather than a dot. We can make this the default behavior using

```
trellis.par.set(box.dot = list(pch = "|"))
```

For one categorical variable, we can use a bar graph.

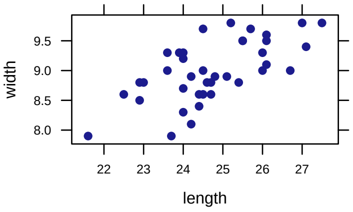
```
bargraph( ~ sex, data=HELPrct)  # categorical variable
```

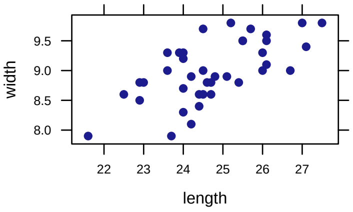


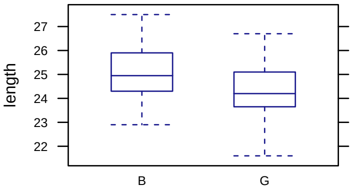
Note
The lattice package does not supply a function for creating pie charts. This is no great loss since it is generally harder to make comparisons using a pie chart.

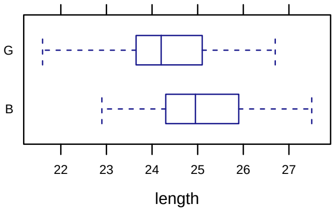
Two-variable plots are also very similar.

```
    xplot( width ~ length, data=KidsFeet)  # 2 quantitative vars
plotPoints( width ~ length, data=KidsFeet)  # mosaic alternative
    bwplot(length ~ sex,    data=KidsFeet)  # 1 cat; 1 quant
    bwplot(   sex ~ length, data=KidsFeet)  # reverse roles
```





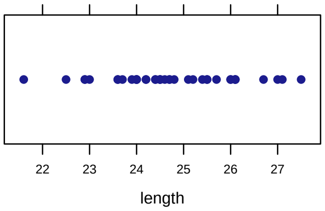


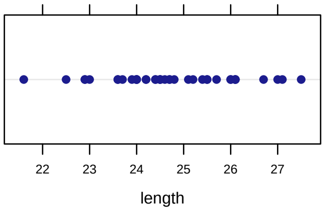


The lattice package also provides the stripplot() and dotplot() functions which can be used for one-dimensional scatter plots. These work reasonably well for small data sets but are of limited utility for larger data sets.

Caution!
There is also a function dotPlot() (with a capital P). Note that dotplot() produces a very different kind of plot from that produced by dotPlot().

```
stripplot( ~ length, data=KidsFeet)
  dotplot( ~ length, data=KidsFeet)
```

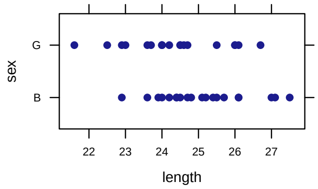


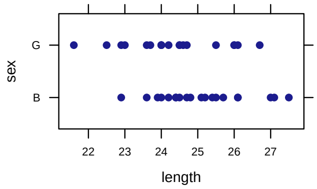


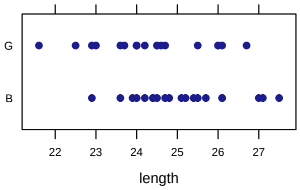
These and xyplot() or plotPoints() can also be used with one quantitative variable and one categorical variable.

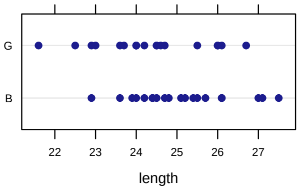
Teaching Tip
We generally don't introduce dotplot() and stripplot() to students but simply use xyplot() or plotPoints().

```
    xyplot(sex ~ length, data=KidsFeet)
plotPoints(sex ~ length, data=KidsFeet)
 stripplot(sex ~ length, data=KidsFeet)
   dotplot(sex ~ length, data=KidsFeet)
```









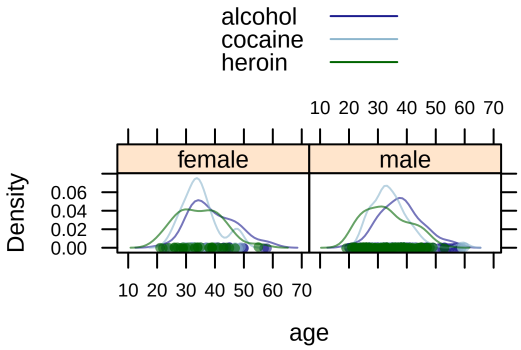
4.2.4 *Groups and sub-plots*

We can add additional variables to our plots either by overlaying multiple plots or by placing multiple plots next to each other in a grid. To overlay plots, we add an extra argument to our template using `groups =` , and to create sub-plots (called **panels** in `lattice` and **facets** in `ggplot2` graphics) using a formula of the form

```
y ~ x | z
```

For example, we can overlay density plots of age for each substance group in separate panels for each sex:

```
densityplot( ~ age | sex, data=HELPrct,
               groups=substance,
               auto.key=TRUE)
```



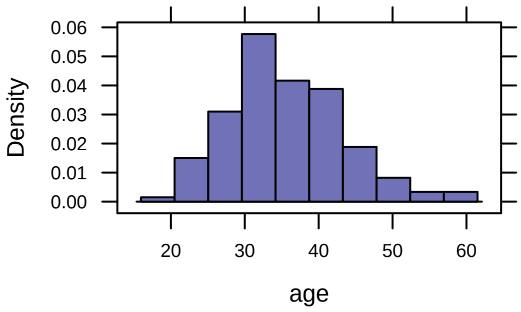
auto.key=TRUE adds a simple legend so we can tell which of the overlaid curves is which.

4.3 Numerical Summaries

Numerical summaries can be created in the same way, we simply replace the plot name with the name of the numerical summary we desire. Nothing else changes; a mean and a histogram each summarise a single variable, so exchanging histogram() for mean() gives us the numerical summary we desire.

NOTE
The important thing to notice in this section is how little there is to learn once you know how to make plots. Simply change the plot name to a summary statistic name and your done.

```
histogram( ~ age, data=HELPrct)
     mean( ~ age, data=HELPrct)

[1] 35.7
```



MORE INFO
To see the full list of these formula-aware numerical summary functions, use help(favstats).

The mosaic package includes formula-aware versions of several numerical summaries, including mean(), sd(), var(), min(), max(), sum(), IQR(). In addition, the favstats() function computes many of our favorite statistics all at once:

```
favstats( ~ age, data=HELPrct)

 min Q1 median Q3 max mean   sd   n missing
  19 30     35 40  60 35.7 7.71 453       0
```

The tally() function can be used to count cases.

```
tally( ~ sex, data=HELPrct)


female   male
   107    346

tally( ~ substance, data=HELPrct)


alcohol cocaine  heroin
    177     152     124
```

Sometimes it is more convenient to display proportions or percents.

```
tally( ~ substance, data=HELPrct, format="percent")


alcohol cocaine  heroin
   39.1    33.6    27.4

tally( ~ substance, data=HELPrct, format="proportion")


alcohol cocaine  heroin
  0.391   0.336   0.274
```

Summary statistics can be computed separately for multiple subsets of a data set. This is analogous to plotting multiple variables and can be thought about in three ways. Each of these computes the same value.

```
# age dependant on substance
sd(   age ~ substance, data=HELPrct)

alcohol cocaine  heroin 
   7.65    6.69    7.99 

# age separately for each substance
sd( ~ age | substance, data=HELPrct)

alcohol cocaine  heroin 
   7.65    6.69    7.99 

# age grouped by substance
sd( ~ age, groups=substance, data=HELPrct)

alcohol cocaine  heroin 
   7.65    6.69    7.99 
```

The favstats() function can compute several numerical summaries for each subset

```
favstats(age ~ substance, data=HELPrct)

  substance min Q1 median   Q3 max mean   sd   n missing
1   alcohol  20 33   38.0 43.0  58 38.2 7.65 177       0
2   cocaine  23 30   33.5 37.2  60 34.5 6.69 152       0
3    heroin  19 27   33.0 39.0  55 33.4 7.99 124       0
```

Similarly, we can create two-way tables that display either as counts or proportions.

```
tally(sex ~ substance, data=HELPrct)

        substance
sex      alcohol cocaine heroin
  female      36      41     30
  male       141     111     94

tally( ~ sex + substance, data=HELPrct)

        substance
sex      alcohol cocaine heroin
  female      36      41     30
  male       141     111     94
```

Marginal totals can be added with `margins=TRUE`

```
tally(sex ~ substance, data=HELPrct, margins=TRUE)

        substance
sex      alcohol cocaine heroin
  female      36      41     30
  male       141     111     94
  Total      177     152    124

tally( ~ sex + substance, data=HELPrct, margins=TRUE)

        substance
sex      alcohol cocaine heroin Total
  female      36      41     30   107
  male       141     111     94   346
  Total      177     152    124   453
```

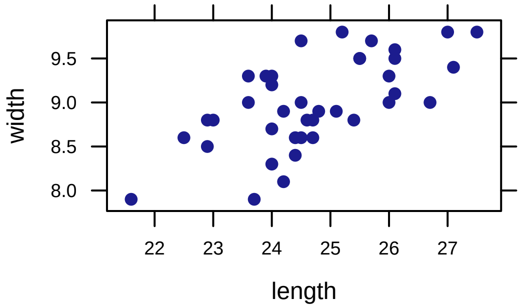
4.4 *Linear models*

Although we have not mentioned linear models yet, they are an important motivation for the template approach to graphical and numerical summaries. The `lattice` graphics system already makes use of the same template as linear models, and the `mosaic` package makes it possible to do numerical summaries with the same template. By introducing students to the template for graphical and numerical summaries, there is very little new to learn when they are ready to fit a model.

Perhaps you are thinking this means that we don't need to wait so long to introduce modeling in the introductory statistics course. We think so too. See the companion volume, *Start Modeling in R*.

For example, suppose we want to know how the width of kids' feet depends on the length of the their feet. We could make a scatter plot and we can construct a linear model using the same template

```
xyplot(width ~ length, data=KidsFeet)
    lm(width ~ length, data=KidsFeet)


Call:
lm(formula = width ~ length, data = KidsFeet)

Coefficients:
(Intercept)       length
      2.862        0.248
```



We'll have more to say about modeling elsewhere. For now, the important point is that our use of the template for graphing and numerical summaries prepares students to ask how does y depend on x and to formalize models of two or more variables when the time comes.

4.5 *A few other tests*

Many introductory statistics classes introduce students to one- and two-sample tests for means and proportions. The `mosaic` package brings these into the template as well.

```
t.test( ~ length, data=KidsFeet)


One Sample t-test

data:  data$length
t = 100, df = 40, p-value <2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 24.3 25.2
sample estimates:
mean of x
     24.7
```

More Info

For a more thorough treatment of how to use R for the core topics of a traditional introductory statistics course, see *A Student's Guide to R*.

More Info

Chi-squared tests can be performed using `chisq.test()`. This function is a little different in that it operates on tabulated data of the sort produced by `tally()` rather than on the data itself. So the use of the template happens inside `tally()` rather than in `chisq.test()`.

The output from these functions also includes more than we really need. The `mosaic` package provides `pval()` and `confint()` for extracting p-values and confidence intervals:

```
 pval(t.test( ~ length, data=KidsFeet))

  p.value
 3.06e-50

 confint(t.test( ~ length, data=KidsFeet))

   mean of x lower upper level
 1      24.7  24.3  25.2  0.95
```

```
confint(t.test(length ~ sex, data=KidsFeet))

  mean in group B mean in group G  lower upper level
1            25.1            24.3 -0.045  1.61  0.95
```

```
# using Binomial distribution
confint(binom.test( ~ sex, data=HELPrct))

  probability of success lower upper level
1                  0.236 0.198 0.278  0.95
```

```
# using normal approximation to the binomial distribution
confint(prop.test( ~ sex, data=HELPrct))

      p lower upper level
1 0.236 0.198 0.279  0.95

confint(prop.test(sex ~ homeless, data=HELPrct))

  prop 1 prop 2  lower    upper level
1  0.191  0.275 -0.165 -0.00143  0.95
```

4.6 *`lattice` bells and whistles*

In the plots we have shown so far, we have focused on creating a variety of useful plots and (for the most part) accepted the default presentation of them. The `lattice` graphics system provides many bells and whistles that can be introduced once the graphics template has been mastered. Optional arguments to the graphics functions can be used to add or modify

- the viewing window
- titles,
- axis labels,
- colors, shapes, sizes, and line types,
- transparency,
- fonts

and many other features of a plot. Our advice is to hold off on such bells and whistles until students ask or an analysis demands them.

4.6.1 *Example: Number of births per day*

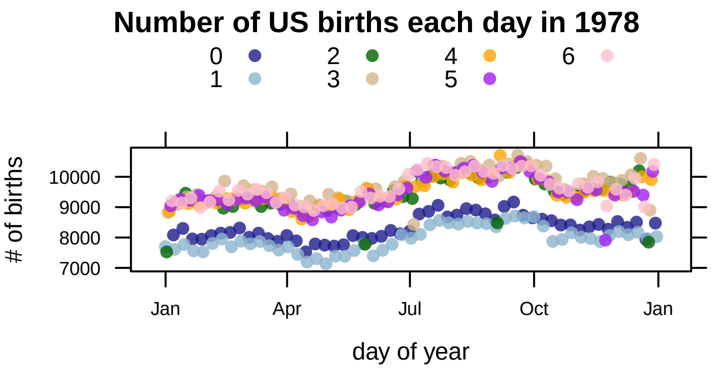
We have seen the `Births78` data set in Section 3.2. The plots below take advantage of additional arguments to improve the plot. The first plot below illustrates one of the important features of this data set – there are usually fewer births on two days of the week and more on the other five. From this we can be quite certain that 1978 began on a Sunday.

More Info
`%%` performs modular arithmetic, in this case giving seven groups, one for each day of the week.

```
xyplot(births ~ date, data=Births78,
       groups=dayofyear %% 7,
       auto.key=list(columns=4),
       main="Number of US births each day in 1978",
       xlab="day of year",
       ylab="# of births",
       par.settings=list(
         superpose.symbol=list(pch=16, cex=.8, alpha=.8))
)
```

More Info
Some of the arguments here use lists. Lists are one of the fundamental “container types” in R. Instructors will benefit from being able to recognize them. We will have more to say about them in Chapter 7.

More Info
We could also use the `wday()` function in the `lubridate` package to compute the weekday directly from `date`.

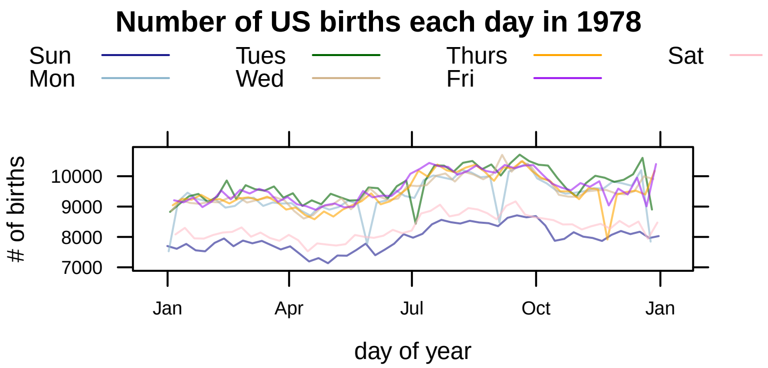


Here we have used

- auto.key to control the layout of the legend (4 columns instead of 1)
- main to set the title for the plot
- xlab and ylab to set the axis labels
- par.settings to set the plot character (pch), character expansion (cex), and opacity (alpha) for overlaid plots (superpose.symbol).

The following plot uses lines instead of points which makes it easier to locate the handful of unusual observations.

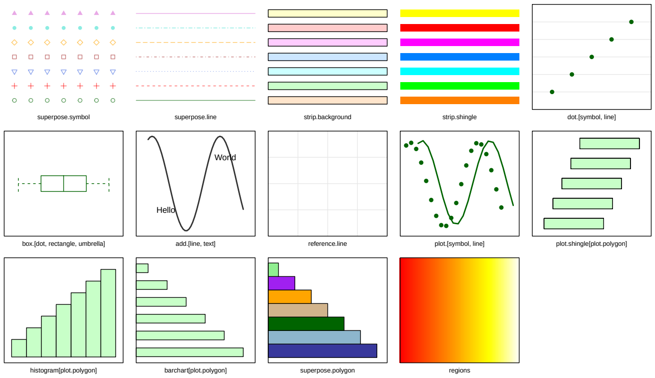
```
xyplot(births ~ date, data=Births78,
  groups=wday, type='l',
  main="Number of US births each day in 1978",
  auto.key=list(columns=4, lines=TRUE, points=FALSE),
  xlab="day of year",
  ylab="# of births"
)
```



4.6.2 *Themes*

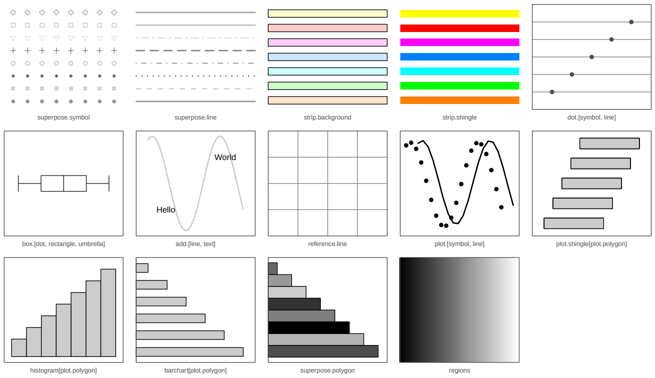
Settings that are used repeatedly can be collected into a theme. The mosaic package provides such a theme called theme.mosaic(). The show.settings() function displays the settings of the currently active theme.

```
trellis.par.set(col.whitebg())
show.settings()
```

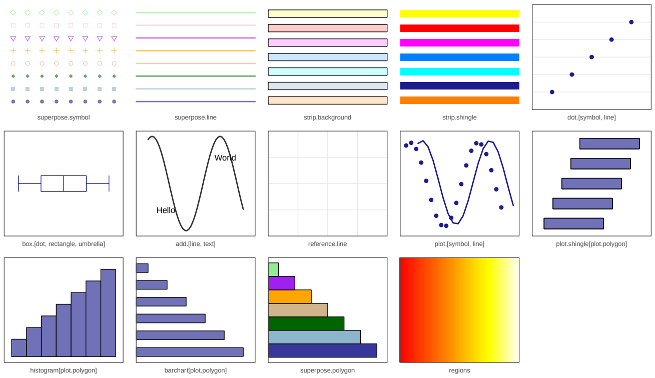


More Info
In the printed version of this book, all three examples appear in black and white and were processed with `theme.mosaic(bw=TRUE)`. In the online version, the first and third examples appear in color.

```
trellis.par.set(theme.mosaic(bw=TRUE))
show.settings()
```

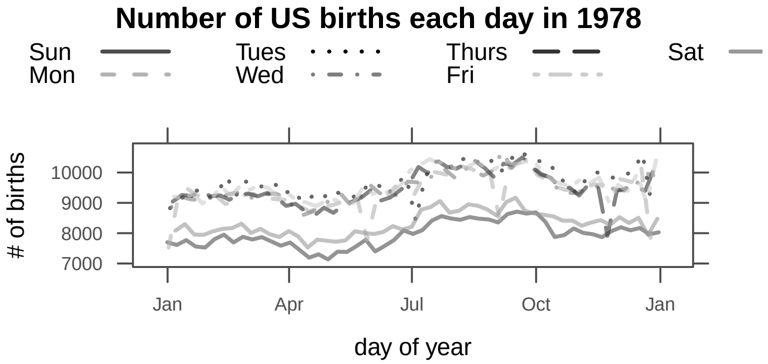


```
trellis.par.set(theme.mosaic())
show.settings()
```



Themes can also be assigned to `par.settings` if we want them to affect only one plot:

```
xyplot(births ~ date, data=Births78,
  groups=wday, type='l',
  main="Number of US births each day in 1978",
  auto.key=list(columns=4, lines=TRUE, points=FALSE),
  par.settings=theme.mosaic(bw=TRUE),
  xlab="day of year",
  ylab="# of births"
)
```



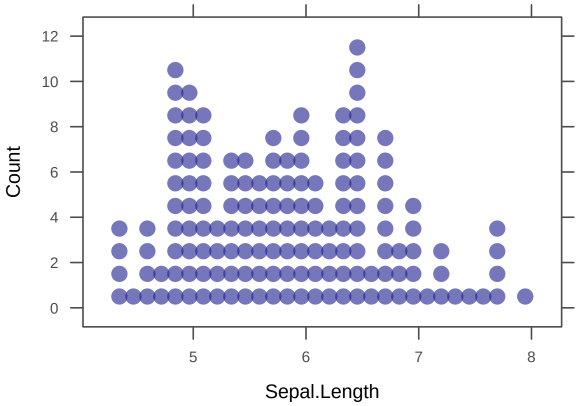
4.7 Some additional examples

4.7.1 Dot plots

Dotplots are not as commonly seen in the statistical literature as they are in statistics education, where they can serve an important role in helping students learn to interpret histograms (and frequency polygons and density plots). A **dot plot** represents each value of a quantitative variable with a dot. The values are rounded a bit so that the dots line up neatly, and dots are stacked up into little towers when the data values cluster near each other. Dot plots are primarily used with modestly sized data sets and can be used as a bridge to the other plots, where there is no longer a direct connection between a component of the plot and an individual observation.

Here is an example using the sepal lengths recorded in the `iris` data set.

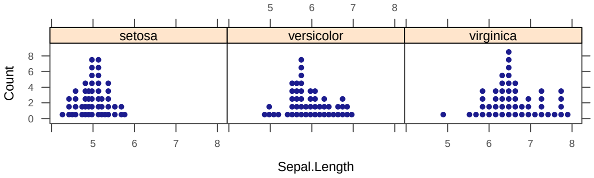
```
dotPlot(~ Sepal.Length, data=iris,
                n=30,      # approx. 30 bins/columns
                alpha=.6)  # partially transparent
```



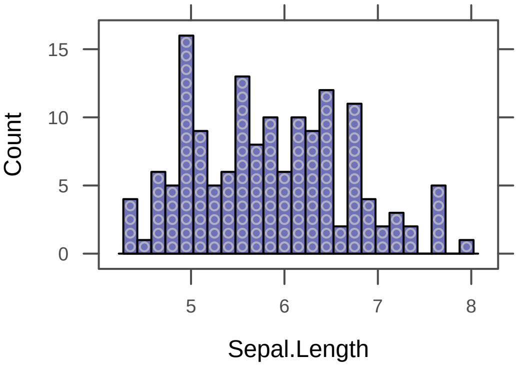
We can use a conditional variable to give us separate dot plots for each of the three species in this data set.

```
dotPlot(~ Sepal.Length | Species, data=iris, n=20,
         layout=c(3,1)) # 3 columns (x) and 1 row (y)
```

Teaching Tip
Dot plots are useful for displaying sampling distributions and bootstrap distributions, especially if the total number of dots is chosen to be something simple like 1000. In that case, probabilities can be easily estimated by counting dots.



The connection between histograms and dot plots can be visualized by overlaying one on top of the other.

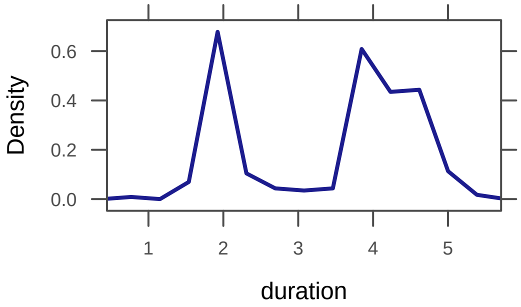


4.7.2 Frequency polygons: `freqpolygon()`

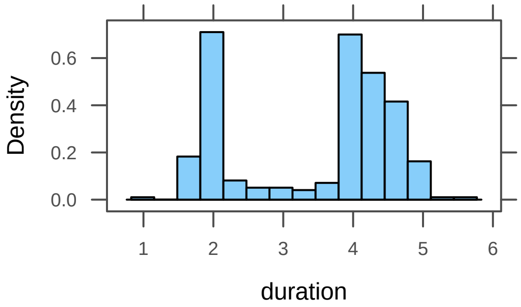
Frequency polygons and **density plots** provide alternatives to histograms that make it easier to overlay the representations of multiple subsets of the data. A frequency polygon is created from the same data summary (bins and counts) as a histogram, but instead of representing each bin with a bar, it is represented by a point (at the center of the where the top of the histogram bar would have been).

These points are then connected with line segments. Here is an example that shows the distribution of Old Faithful eruptions times from a sequence of observations

```
require(MASS)
freqpolygon( ~ duration, data=geyser, n=15)
```

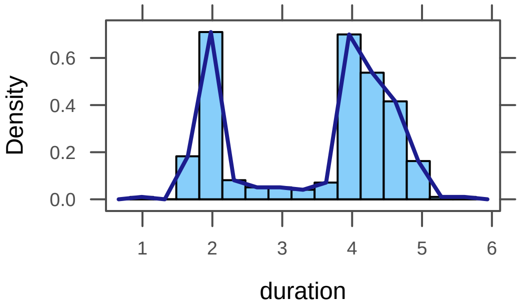


Numerically, the data are being summarized and represented in exactly the same way as for histograms, but visually the horizontal and vertical line segments of the histogram are replaced by sloped line segments.



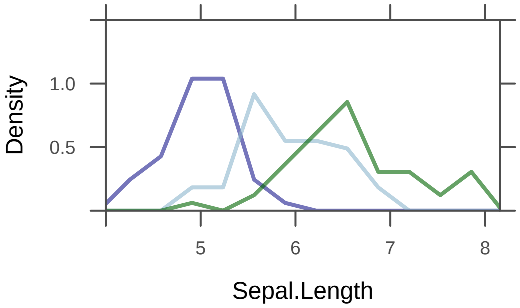
CAUTION!
The `faithful` data set contains similar data, but the variable names in that data frame are poorly chosen. The `geyser` data set in the `MASS` package has better names and more data.

TEACHING TIP
Point out that an interesting feature of this distribution is its clear bimodality. In particular, the mean and median eruption time are not a good measures of the duration of a "typical" eruption since almost none of the eruption durations are near the mean and median.



This may give a more accurate visual representation in some situations (since the distribution can "taper off" better). More importantly, it makes it much easier to overlay multiple distributions.

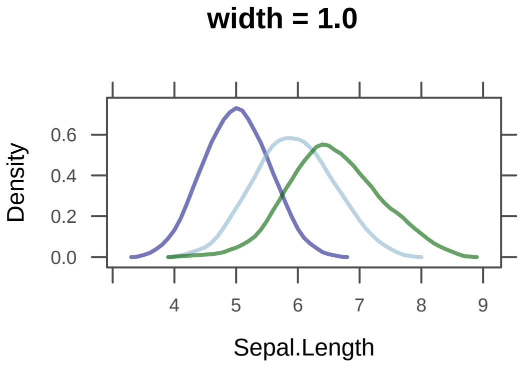
```
freqpolygon( ~ Sepal.Length, data=iris,
                        groups=Species,
                        ylim=c(0,1.5)  # manually set y-axis range
)
```

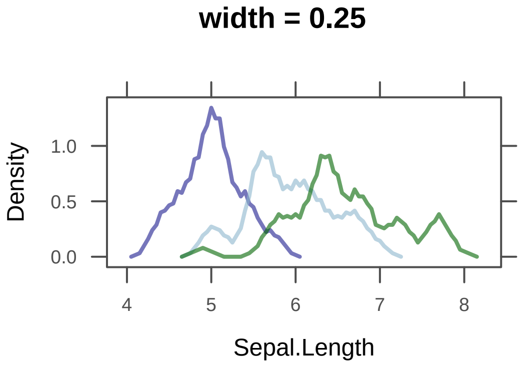


4.7.3 *ASH plots: Average Shifted Histograms*

Histograms are sensitive to the choice of bin widths and edges (or centers). One way to reduce this dependency is called an Average Shifted Histogram or ASH plot. The height of an ASH plot is the average height over all histograms of a fixed bin width. If you are familiar with density plots (discussed in the next section), ASH plots will remind you them, but they are far easier to explain to beginners.

```
ashplot( ~ Sepal.Length, data=iris, groups=Species,
         width = 1.0, main = "width = 1.0")
ashplot( ~ Sepal.Length, data=iris, groups=Species,
         width = 0.25, main = "width = 0.25")
```

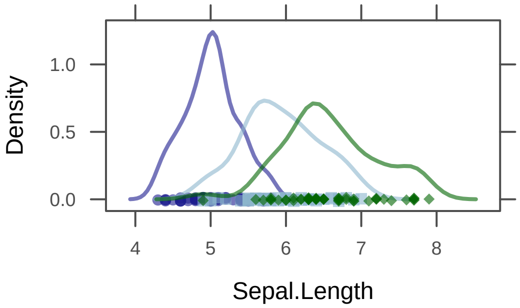




4.7.4 *Density plots:* `densityplot()`

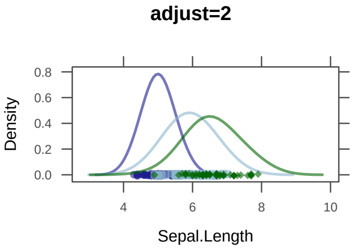
Density plots are similar to frequency polygons, but the piecewise linear representation is replaced by a smooth curve.

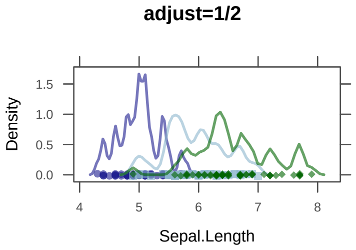
```
densityplot( ~ Sepal.Length, data=iris, groups=Species)
```



Beginners do not need to know the details of how that smooth curve is generated, but should be introduced to the `adjust` argument which controls the degree of smoothing. It is roughly equivalent to choosing wider or narrower bins for a histogram or frequency polygon. The default value is 1. Higher values smooth more heavily; lower values, less so.

```
densityplot( ~ Sepal.Length, data=iris, groups=Species,
             adjust=3, main="adjust=2")
densityplot( ~ Sepal.Length, data=iris, groups=Species,
             adjust=1/3, main="adjust=1/2")
```





4.7.5 *The Density Scale*

There are three scales that can be used for the plots in the preceding section: `count`, `percent`, and `density`. Beginning students will be most familiar with the `count` scale and perhaps also the `percent` scale, but most will not have seen the `density` scale. The density scale captures the most important aspect of all of these plots:

Area is proportional to frequency.

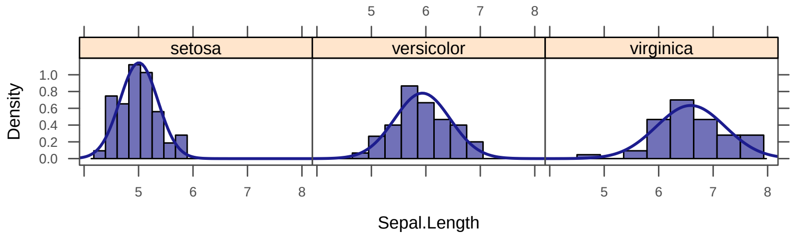
The density scale is chosen so that the constant of proportionality is 1, in which case we have

Area equals proportion.

Teaching Tip
Create some histograms or frequency polygons with a density scale and see if your students can determine what the scale is. Choosing convenient bin widths (but not 1) and comparing plots with different bin widths and different scale types can help them reach a good conjecture about the density scale.

This is the only scale available for `densityplot()` and is the most suitable scale if one is primarily interested in the *shape* of the distribution. The vertical scale is affected very little by the choice of bin widths or `adjust` multipliers. It is also the appropriate scale to use when overlaying a density function onto a histogram, something the `mosaic` package makes easy to do.

```
histogram( ~ Sepal.Length | Species, data=iris, fit="normal")
```

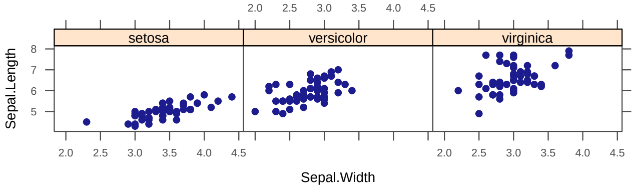


The other scales are primarily of use when one wants to be able to read off bin counts or percents from the plot.

4.7.6 Groups or panels?

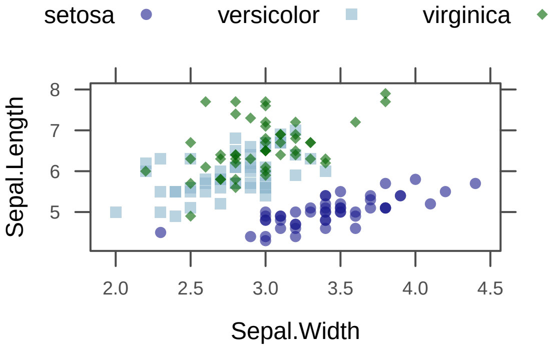
The following examples using the `iris` data set provide a comparison of using groups or panels to separate subsets of the data. First we put the three species into three separate panels.

```
xyplot(Sepal.Length ~ Sepal.Width | Species, data=iris,
  layout=c(3,1)) # layout controls number of columns and rows
```



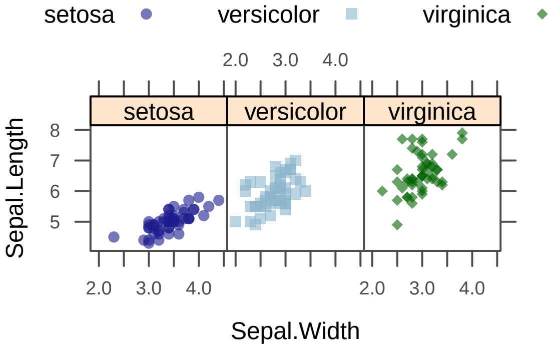
Alternatively, we can use the `groups` argument to indicate the different species using different symbols on the same panel.

```
xyplot(Sepal.Length ~ Sepal.Width, groups=Species,
  auto.key=list(columns=3), data=iris)
```



Sometimes it is helpful to use both panels and symbol `groups`.

```
xyplot(Sepal.Length ~ Sepal.Width | Species, groups=Species,
  auto.key=list(columns=3), data=iris)
```



4.7.7 Dealing with long labels

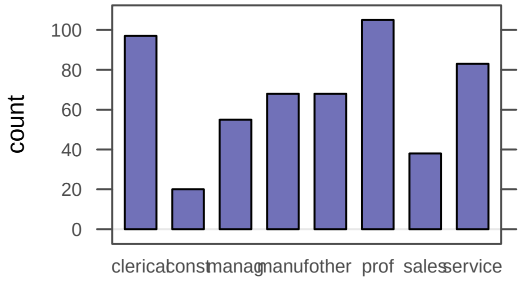
Suppose we want to display the following table (based on data from the *1985 Current Population Survey*) using bar graph.

```
tally( ~ sector, data=CPS85)
```

```
clerical    const    manag    manuf    other     prof 
      97       20       55       68       68      105 
   sales  service 
      38       83 
```

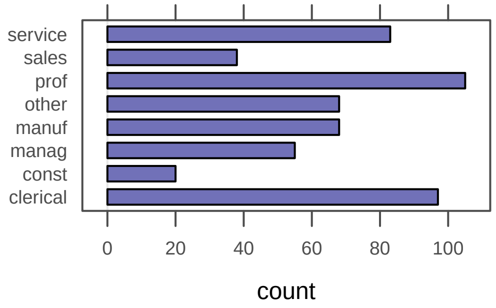
The mosaic function bargraph() can display these tables as bar graphs, but there isn't enough room for the labels.

```
bargraph(~ sector, data=CPS85)
```



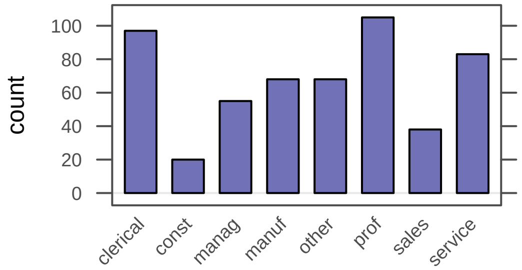
One solution would be to use horizontal bars

```
# horizontal bars
bargraph(~ sector, data=CPS85, horizontal=TRUE)
```



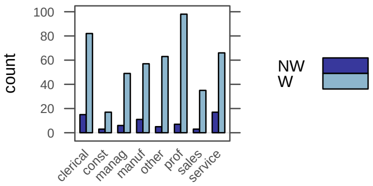
Another is to rotate the labels.

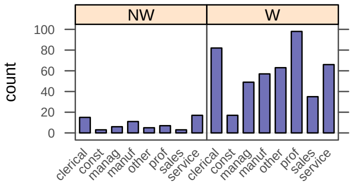
```
bargraph(~ sector, data=CPS85,
                   scales=list(x=list(rot=45)))
```



As with the other lattice plots, we can add grouping or conditioning to our plot.

```
bargraph(~ sector, data=CPS85, groups=race,
                   auto.key=list(space="right"),
                   scales=list(x=list(rot=45)))
bargraph(~ sector | race, data=CPS85,
                          scales=list(x=list(rot=45)))
```





4.8 *Saving Your Plots*

There are several ways to save plots in RStudio, but the easiest is probably the following:

1. In the Plots tab, click the "Export" button.
2. Copy the image to the clipboard using right click.
3. Go to your document (e.g. Microsoft Word) and paste in the image.
4. Resize or reposition your image as needed.

You can save all of this exporting and copying and pasting if you use RMarkdown, or `knitr`/LaTeX to prepare your documents.

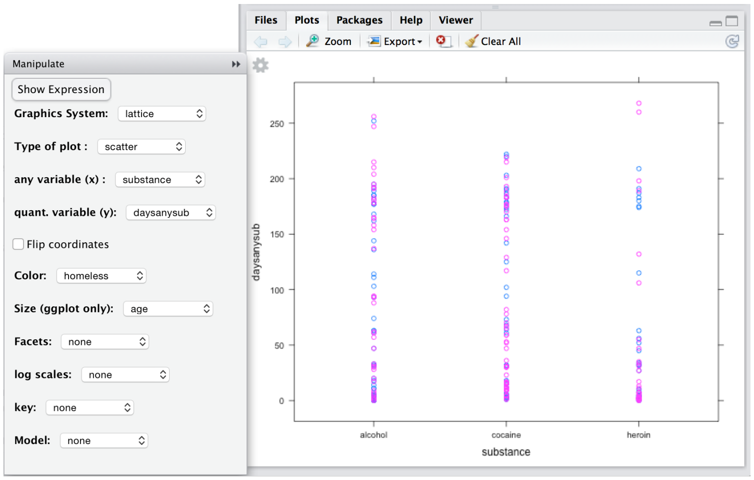
Altenatively, a plot can be exported to a file.

R also provides function like pdf() and png() that can be used to save plots in a varity of formats. See the documentation of these functions for details and links to functions that can be used to save graphics in other file formats.

4.9 mplot()

The mplot() function does a number of different things, depending on what information it is provided. When mplot() is given a data frame in RStudio, it opens up an interactive plot with controls that allow the user to select variables and create plots of various sorts.

More Info
mplot() is a *generic* function. R includes many generic functions (like print() and plot() and summary()). These functions inspect the objects passed as arguments (at least the first one) and decide what to do based on the class of the argument(s).



Manipulate

Show Expression

Graphics System: lattice

Type of plot : scatter

any variable (x) : substance

quant. variable (y): daysanysub

□ Flip coordinates

Color: homeless

Size (ggplot only): age

Facets: none

log scales: none

key: none

Model: none

The plots can be made using lattice or ggplot2, and there is a "Show expression" button that displays the code used to create the plot. This can be used to learn how to make the plot and can be copied and pasted into the console or documents.

The use of mplot() makes it easy to explore a number of plots quickly and can facilitate learning either lattice or ggplot2 by showing the code used to create the plots.

Caution!
This feature of mplot() takes advantage of the manipulate package and so works only within RStudio. See Chapter 8 for more about manipulate.

4.10 Review of R Commands

Here is a brief summary of the commands introduced in this chapter.

```
require(mosaic)                          # load the mosaic package
require(mosaicData)                      # load the mosaic data sets

tally( ~ sector, data=CPS85)             # frequency table
tally( ~ sector + race, data=CPS85)      # cross tabulation of sector by race
mean( ~ age, data = HELPrct)             # mean age of HELPrct subjects
mean( ~ age | sex, data = HELPrct)       # mean age of male and female subjects
mean(age ~ sex, data = HELPrct)          # mean age of male and female subjects
median(x); var(x); sd(x);                # more numerical summaries
quantile(x); sum(x); cumsum(x)           # still more summaries
favstats( ~ Sepal.Length, data=iris)     # compute favorite numerical summaries

histogram( ~ Sepal.Length | Species, data=iris)     # histograms (with extra features)
dotPlot( ~ Sepal.Length | Species, data=iris)       # dot plots for each species
freqpolygon( ~ Sepal.Length, groups = Species, data=iris) # overlaid freq. polygons
densityplot( ~ Sepal.Length, groups = Species, data=iris) # overlaid densityplots
qqmath( ~ age | sex, data=CPS85)                          # quantile-quantile plots
bwplot(Sepal.Length ~ Species, data = iris)               # side-by-side boxplots
xyplot(Sepal.Length ~ Sepal.Width | Species, data=iris)   # side-by-side scatter plots
bargraph( ~ sector, data=CPS85)                           # bar graph



mplot(HELPrct)                         # interactive plot (RStudio only)
```

4.11 Exercises

4.1 The `Utilities2` data set in the `mosaic` package contains information about the bills for various utilities at a residence in Minnesota collected over a number of years. Since the number of days in a billing cycle varies from month to month, variables like `gasbillpday` (`elecbillpday`, etc.) contain the gas bill (electric bill, etc.) divided by the number of days in the billing cycle.

a) Use the documentation to determine what the `kwh` variables contains.

b) Make a scatter plot of `gasbillpday` vs. `monthsSinceY2K` using the command

```
xyplot(gasbillpday ~ monthsSinceY2K, data=Utilities2,
                     type='l')        # the letter l
```

What pattern(s) do you see?

c) What does `type='l'` do? Make your plot with and without it. Which is easier to read in this situation?

d) What happens if we replace `type='l'` with `type='b'`?

e) Make a scatter plot of `gasbillpday` by `month`. What do you notice?

f) Make side-by-side boxplots of `gasbillpday` by `month` using the `Utilities2` data frame. What do you notice?

Your first try probably won't give you what you expect. The reason is that month is coded using numbers, so `R` treats it as numerical data. We want to treat it as categorical data. To do this in `R` use `factor(month)` in place of `month`. `R` calls categorical data a **factor**.

g) Make any other plot you like using this data. Include both a copy of your plot and a discussion of what you can learn from it.

5
Simulation-Based Inference

Resampling approaches have become increasingly important in statistical education[1,2]. The `mosaic` package provides simplified functionality to support teaching inference based on randomization tests and bootstrap methods. Our goal is to focus attention on the important parts of these techniques (e.g., where randomness enters in and how to use the resulting distribution) while hiding some of the technical details involved in creating loops and accumulating values.

5.1 Staring Early

One of the advantages of simulation-based inference is that one can start teaching inference early in the course. Section 3.1 describes an example (based on Fisher's lady tasting tea) that we have often used on the first day of class. Textbooks that use a simulation-based approach also begin their discussion of the inference process immediately, using other examples.[3,4] Even when teaching a more traditional course, simulation of the lady tasting tea or some other example can be introduced early in the course to help students begin to understand the key ideas involved in hypothesis testing and estimation.

5.2 Hypothesis Tests

Hypothesis testing can be thought of as a 4-step process:

1. State the null and alternative hypotheses.

[1] N. Tintle, B. Chance, G. Cobb, S. Roy, T. Swanson, and J. VanderStoep. Combating anti-statistical thinking using simulation-based methods throughout the undergraduate curriculum. *The American Statistician*, 69(4), 2015

[2] Tim C. Hesterberg. What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum. *The American Statistician*, 2015

[3] Robin H Lock, Patti Frazer Lock, and Kari Lock Morgan. *Statistics: Unlocking the Power of Data*. Wiley Global Education, 2012

[4] Nathan Tintle, Beth Chance, George Cobb, Allan Rossman, Soma Roy, Todd Swanson, and Jill VanderStoep. *Introduction to Statistical Investigations*. Wiley Global Education, 2015

2. Compute a test statistic.

3. Determine the p-value.

4. Draw a conclusion.

In a traditional introductory statistics course, once this general framework has been mastered, the main work for students is in applying the correct formula to compute the standard test statistics in step 2 and using a table or computer to determine the p-value based on the known (usually approximate) theoretical distribution of the test statistic under the null hypothesis.

In a simulation-based approach, steps 2 and 3 change. In Step 2, it is no longer required that the test statistic be normalized to conform with a known, named distribution. Instead, natural test statistics, like the difference between two sample means

$$\overline{y}_1 - \overline{y}_2$$

can be used instead of the standard two-sample t test statistic

$$\frac{\overline{y}_1 - \overline{y}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} .$$

In Step 3, we use randomization to approximate the sampling distribution of the test statistic. Our lady tasting tea example demonstrates how this can be done from first principles as early as the first day of class.[5] This example is a bit unusual, however. Because the sampling distribution is so simple, the simulation required to create a randomization distribution is completely specified without reference to the data: It's a binomial distribution with parameters determined by the sample size and the null hypothesis, and we can simulate it with `rflip()`.

More typically, we will use randomization to create new simulated data sets that are like our original data in some ways, but make the null hypothesis true. For each simulated data set, we calculate our test statistic, just as we did for the original sample. Together, this collection of test statistics computed from the simulated samples constitute our randomization distribution.

When creating a randomization distribution, we will attempt to satisfy 3 guiding principles.

[5] See Section 3.1.

1. Be consistent with the null hypothesis.

   We need to simulate a world in which the null hypothesis is true. If we don't do this, we won't be testing our null hypothesis.

2. Use the data in the original sample.

   The original data should shed light on some aspects of the distribution that are not determined by null hypothesis. For example, a null hypothesis about a mean doesn't tell us about the shape of the population distribution, but the data give us some indication.

3. Reflect the way the original data were collected.

5.2.1 *Permutations tests using shuffle()*

The mosaic package provides shuffle() as a synonym for sample(). When used without additional arguments, this will permute its first argument.

```
shuffle(1:10)

 [1]  3  8  4  7  6  1 10  9  2  5

shuffle(1:10)

 [1] 10  5  6  9  1  7  8  4  3  2
```

Applying shuffle() to an explanatory variable allows us to test the null hypothesis that the explanatory variable has, in fact, no explanatory power. This idea can be used to test

- the equivalence of two or more proportions,
- the equivalence of two or more means,
- whether a regression parameter is 0.

For example, let's test whether young men and women have the same mean body temperature using a data set that contains body temperatures for 50 college students, 25 men and 25 women.

```
require(Lock5withR)
inspect(BodyTemp50)
```

```
categorical variables:
  name  class levels  n missing
1  Sex factor      2 50       0
                                     distribution
1 Female (50%), Male (50%)

quantitative variables:
      name   class  min   Q1 median   Q3 max mean    sd  n
1 BodyTemp numeric 96.4 97.8   98.2 98.8 101 98.3 0.765 50
2    Pulse integer 57.0 70.2   75.0 78.0  89 74.4 6.440 50
3   Gender integer  0.0  0.0    0.5  1.0   1  0.5 0.505 50
  missing
1       0
2       0
3       0
```

1. State the null and alternaive hypotheses.

   - H_0: mean body temperature is the same for males and females.
   - H_a: mean body temperature differs between males and females.

2. Compute a test statistic.

```
favstats( BodyTemp ~ Sex, data = BodyTemp50)
```

```
     Sex  min   Q1 median   Q3   max mean    sd  n missing
1 Female 96.9 97.6   98.1 98.7  99.5 98.2 0.675 25       0
2   Male 96.4 98.0   98.3 98.8 100.8 98.3 0.851 25       0
```

```
T <- diffmean( BodyTemp ~ Sex, data = BodyTemp50); T
```

```
diffmean
   0.176
```

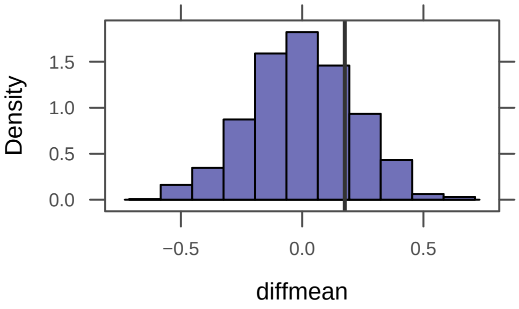
3. Use randomiztion to compute a p-value.

```
Temp2.Null <-
  do(1000) * diffmean( BodyTemp ~ shuffle(Sex), data = BodyTemp50)
histogram( ~ diffmean, data = Temp2.Null, center = 0, v = 0.176)
tally( ~ (diffmean >= T), data = Temp2.Null)
```

```
 TRUE FALSE
  214   786
```

```
prop( ~ (diffmean >= T), data = Temp2.Null)
```

```
 TRUE
0.214
```



4. Draw a conclusion.

   The p-value is large, so these data offer no reason to reject the hpythesis that male and female college students have the same mean body temperature.

5.2.2 *Computing p-values*

In the preceding example, we hardly needed to compute a p-value because the histogram clearly showed that the observed test statistic (0.176) would not be unusual even if the null hypothesis were true, so these data don't offer any reason to reject the null hypothesis that male and female college students have the same mean body temperature.

Nevertheless, there are two issues related to p-value calculations that we want to address with this example: including the observed test statistic in the null distribution, and calculating 2-sided p-values.

Caution!
If you are using a text book that covers randomization tests, you will need to check whether they include the test statistic computed from the original data in the null distribution or not.

If the null hypothesis is true, then not only our randomly generated data, but also the original data were generated in a world in which the null hypothesis is true. So it makes sense to add the original test statistic to the randomization distribution before calculating the p-value. This has two advantages. First, it ensures that our type I error rate is no larger than the nominal rate. Second, it avoids reporting a p-value of 0 since there will always be at least one test statistic at least as extreme as the one computed from the original data, namely the one computed from the original data.

To simplify this calculation, we may choose to use 999 or 9999 replicates instead of 1000 or 10,000. The `mosaic` package also includes the `prop1()` function which adds an additional count to both the numerator and denominator for the purpose of automating this sort of p-value calculation. This will result in a slightly larger (one-sided) p-value.

Caution!
Although using 999 or 9999 replicates results in p-values that are “round numbers", there is some risk that students will use the 999 vs 1000 distinction as their primary way to tell whether you are creating a randomization dsitribution or a bootstrap distribution.

```
prop1( ~ (diffmean >= T), data = Temp2.Null)

 TRUE
0.215
```

The only challenge for the instructor is to decide if and when to introduce this minor change to the p-value calculation.

But we need a two-sided p-value given our alternative hypothesis. The preferred way to calculate 2-sided p-values is also the simplest: just double the 1-sided p-value.

```
2 * prop1( ~ (diffmean >= T), data = Temp2.Null)

TRUE
0.43
```

An alternative approach sometimes seen would add the proportion of the randomization distribution that is below $-T = -0.176$. For a symmetric randomization distribution, this should give a very similar result, but it does not perform as well when the randomization distribution is skewed, is slightly more difficult to compute, and is not transformation invarient, so tests that are equivalent as 1-sided tests might not result in equivalent 2-sided tests. It seems there is no reason to introduce this method to students.

Teaching Tip
But this alternative might be covered in the text book you are using, so students might use it even if you don't teach it.

5.2.3 *Some additional examples*

The technique of shuffling an explanatory variable can be applied to a wide range of situations. The following templates illustrate the similarity among these.

```
Two.Proportions <- do(999) * diffprop(y ~ shuffle(x),     data = Data)
Two.Means       <- do(999) * diffmean(y ~ shuffle(x),     data = Data)
Linear.model    <- do(999) *       lm(y ~ shuffle(x) + a, data = Data)
Two.Way.Table   <- do(999) *    chisq(y ~ shuffle(x),     data = Data)
```

As an example, let's consider the proportion of subjects in the Health Evaluation and Linkage to Primary Care who were admitted to the substance abuse program for each of three substances: alcohol, cocaine, and heroin. We'd like to know if there is evidence that these proportions differ for men and for women. In our data set, we observe modest differences.

Note
The chisq() function computes the chi-squared statistic either from a formula and data frame, from a table produced by tally(), or from an object produced by chisq.test().

```
tally( substance ~ sex, data = HELPrct,
       format="prop", margins = TRUE)

         sex
substance female  male
  alcohol  0.336 0.408
  cocaine  0.383 0.321
  heroin   0.280 0.272
  Total    1.000 1.000
```

Could those differences be attributed to chance? Or do these results provide reliable evidence that the drug of choice varies (a bit) between men and women?

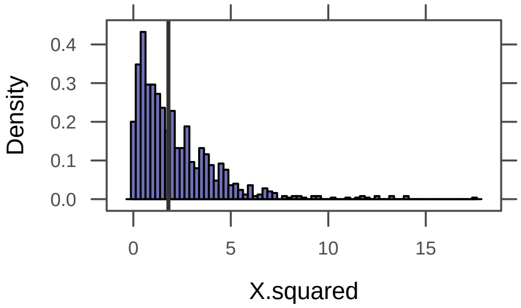
We can simulate a world in which the proportions vary only because of random sampling variability using shuffle() to permute the sex (or equivalently substance) labels.

```
T <- chisq(substance ~ shuffle(sex), data = HELPrct); T  # test statistic
```

```
X.squared
      1.8
```

```
Substance.Null <-
  do(999) * chisq(substance ~ shuffle(sex), data = HELPrct)
histogram( ~ X.squared, data = Substance.Null, v = T, width = 0.25)
prop1( ~(X.squared >= T), data = Substance.Null)
```

```
 TRUE
0.453
```



Both the histogram and our randomization p-value suggest that the differences observed between men and women are not statistically significant.

5.2.4 *Testing a single mean*

One wrinkle in our system is the test for a single mean. Let's illustrate with a test of $H_0 : \mu = 98.6$ using our sample of 50 body temperatures. Testing a null hypothesis of the form

- H_0: $\mu = \mu_0$

is a bit of a special case. Unlike the examples above, there is no explanatory variable to shuffle. Unlike a test for a single proportion, the null hypothesis does not completely specify the sampling distribution.

At least there is an obvious candidate for a test statistic: the sample mean, $\overline{y}$.

NOTE
Somewhat surprisingly, this is the most challenging hypothesis test to handle with our system. See below for one reason this doesn't bother us too much.

NOTE
Many books use $\overline{x}$ here instead of $\overline{y}$.

```
mean( ~ BodyTemp, data = BodyTemp50)
```

```
[1] 98.3
```

This test statistic is easily applied to any data set, we just need a way to generate random data sets in which the null hypothesis is true. As mentioned above, there is no explanatory variable to shuffle. If we shuffle `BodyTemp` (or the entire data set), we will get the same mean every time, since the mean does not depend on order.

Instead, we sample this time with replacement. The `resample()` function does this.

```
resample(1:10)   # notice the duplicates
```

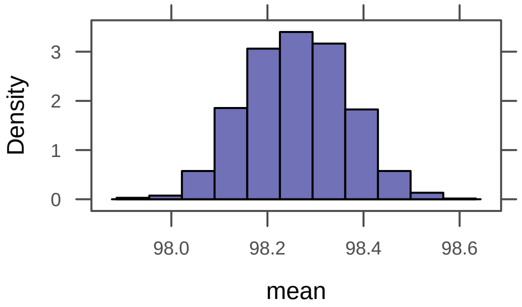
```
 [1] 10  5  5  6  1  3  8  4  6  2
```

We can resample individual variables or the entire data frame. (Since there is only one variable involved in this analysis, the results would be essentially the same either way.)

```
# this doesn't work:
Temp0.Null <-
  do(999) * mean( ~ BodyTemp, data = resample(BodyTemp50))
```

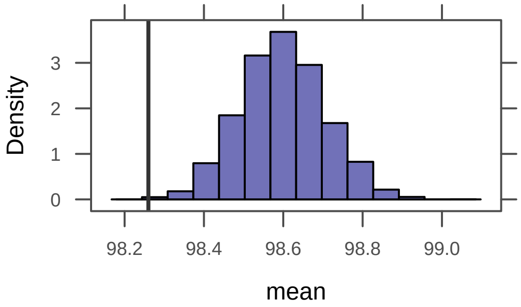
Unfortunately, `Temp0.Null` is not a randomization distribution. Inspecting a histogram shows that the distribution is not centered at 98.6, so we are not simulating a world in which the null hypothesis is true.

```
histogram( ~mean, data = Temp0.Null)
```



Instead it is centered at the mean of our original sample, 98.26. This hints at a way to create a proper randomization distribution. We can shift the distribution by $98.6 - 98.26 = 0.34$. That will result in a distribution that has the same shape as our data but a mean of 98.6, as the null hypothesis demands.

```
Temp1.Null <- do(9999) *
  mean( ~ BodyTemp + (98.6 - 98.26), data = resample(BodyTemp50))
histogram( ~ mean, data = Temp1.Null, v = 98.26, center = 98.6)
```



As before, we can now estimate a p-value by tallying how often we see a value at least as small as 98.26.

```
2 * prop1( ~ (mean <= 98.26), data = Temp1.Null)

  TRUE
0.0016
```

This time the p-value is quite small – it would seem that 98.6 is not the mean body temperature.

Note
We used more replicates in this example to give us a better estimate of this small p-value.

Of all the randomization distributions, randomization distributions used to test hypotheses about a mean are the most awkward to create because of the shifting that is required to center the distribution and the use of `resample()` (which can cause confusion with bootstrap distributions). Fortunately, creating a confidince interval from a bootstrap distribution in this situation is straightforward, and we typically prefer confidence intervals to p-values in this situation.

5.3 *The Bootstrap*

The bootstrap is a method used (primarily) for creating confidence intervals. The basic idea is quite simple and helps reinforce important ideas about what a confidence interval is.

5.3.1 The idea behind the bootstrap

Suppose we want to estimate the mean body temperature using the BodyTemp50 data set. It is simple enough to compute the mean from our data.

```
mean( ~ BodyTemp, data = BodyTemp50)
```

```
[1] 98.3
```

What is missing is some sense for how precise this estimate is. The most common way to present this information is with a confidence interval.

If we had access to the entire population, we could generate many random samples to see how much variability there is in estimates from sample to sample (see Section 5.8). In practice, we will never have access to the entire population (or we wouldn't need to be making estimates). The key idea of the bootstrap is to treat our sample as an approximate representation of the population, and to generate an approximate sampling distribution by sampling (with replacement) *from our sample*. The shape of the bootstrap distribution indicates how precise our estimate is.

Caution!
There are more complicated methods for computing bootstrap confidence intervals that have better performance. We introduce bootstrap confidence intervals using the two simple methods here. Sometimes we return later in the course to talk about the bootstrap-t intervals.

Note
We can use bootstrap methods to estimate the bias in the estimate as well.

Before we proceed, there are a few important things to note about this process.

1. Resampling does not provide a better estimate.

   Resampling is only used to estimate the sample-to-sample *variability* in our estimate, not in an attempt to improve the estimate itself. If we attempted to improve our estimate using our bootstrap samples, we would just make things worse by producing an estimate of our estimate and essentially doubling any bias in the estimation.

2. Resampling works better with large samples than with small samples.

   Small samples are unlikely to represent the population well. While resampling can provide methods that work as well as the traditional methods in standard situations and which can be applied in a wider range of situations without degraded performance, they do not fundamentally alter the need to have a sufficient sample size.

3. The two bootstrap methods we present below are chosen for simplicity, not for performance.

   The primary value in introducing bootstrapping in introductory courses is pedagogical, not scientific. The percentile and standard error intervals introduced below are readily accessible to students and can be applied in a wide range of situations. But they are not the state of the art. In Section 5.7 we will briefly discuss the bootstrap-t interval, a more accurate bootstrap method. Other methods, such as BCa (bias corrected and accelerated) or ABC (approximate bootstrap confidence) also improve upon the percentile and standard error methods, but are beyond the scope of most introductory courses.

   Packages like `resample` and `boot` provide functions for computing intervals using more sophisticated methods.

5.3.2 *Bootstrap confidence intervals for a mean*

Creating a randomization distribution to test a hypothesis about a single mean had some extra challenges. Fortunately, a confidence interval is often preferable in this situation, and creating a bootstrap distribution for a single mean is straightforward: we simply compute the mean body temperature from many resampled versions of our original data.

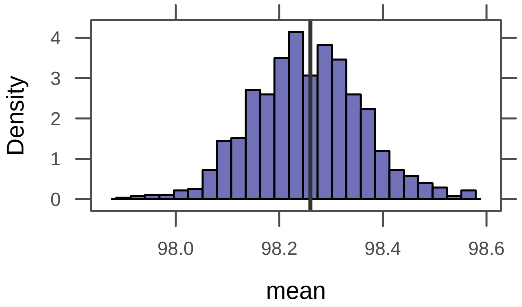
```
Temp.Boot <-
  do(1000) * mean( ~BodyTemp, data = resample(BodyTemp50))
```

When applied to a data frame, the `resample()` function samples rows with replacement to produce a new data frame with the same number of rows as the original, but some rows will be duplicated and others missing.

Ideally, a bootstrap distribution should be unimodal, roughly symmetric, and centered at the original estimate.

CAUTION!
In less than ideal situations, we may need to adjust for bias or use more sophisticated methods. It is good for students to be in the habit of checking these features of the bootstrap distribution before using the simple bootstrap methods we present in this section.

```
mean( ~ BodyTemp, data = BodyTemp50)

[1] 98.3

mean( ~ mean, data = Temp.Boot)

[1] 98.3

histogram( ~ mean, data = Temp.Boot, nint = 25,
           v = mean( ~ BodyTemp, data = BodyTemp50),
           c = mean( ~ BodyTemp, data = BodyTemp50)
           )
```



To compute a 95% percentile confidence interval, we determine the range of the central 95% of the bootstrap distribution. The `cdata()` function automates this calculation.

```
cdata( ~ mean, data = Temp.Boot, p = 0.95)

      low        hi central.p
    98.06     98.48      0.95
```

Alternatively, `qdata()` can be used to obtain the left and right endpoints separately (or for 1-sided confidence intervals).

```
qdata( ~ mean, data = Temp.Boot, p = 0.025)

       p quantile
   0.025   98.058

qdata( ~ mean, data = Temp.Boot, p = 0.975)

       p quantile
   0.975   98.476
```

A second simple method for computing a confidence interval from a bootstrap distribution involves using the boostrap distribution to estimate the standard error.

```
SE <- sd( ~ mean, data = Temp.Boot); SE
```

```
[1] 0.106
```

```
estimate <- mean( ~ BodyTemp, data = BodyTemp50)
estimate
```

```
[1] 98.3
```

```
estimate + c(-1,1) * 2 * SE
```

```
[1] 98.0 98.5
```

This method does not perform as well as the percentile method, but can serve as a good bridge to the formula-based intervals often included even in a course that focuses on simulation-based methods. How to replace the constant 2 with an appropriate value to create more accurate intervals or to allow for different confidence levels is a matter of some subtlety. The simplest method is to use quantiles of a normal distribution, but this will undercover. Replacing the normal distribution with an appropriate t-distribution will widen intervals and can improve coverage, but the t-distribution is only correct in a few cases – such as when estimating the mean of a normal population – and can perform badly when the population is skewed.[6]

Because each of these methods produces a confidence interval that depends only on the distribution of the estimates computed from the resamples, they are easily implemented in wide variety of situations. Calculating either of these simple confidence intervals from the bootstrap distribution can be further automated using an extension to `confint()`.

[6] Tim C. Hesterberg. What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum. *The American Statistician*, 2015

```
confint(Temp.Boot, method = c("percentile", "stderr"))

  name lower upper level     method estimate
1 mean  98.1  98.5  0.95 percentile     98.3
2 mean  98.0  98.5  0.95     stderr     98.3
  margin.of.error df
1              NA NA
2           0.213 49
```

All that remains then is the generation of the bootstrap distribution itself.

5.3.3 *Bootstrap confidence intervals for the difference in means*

If we are interested in a confidence interval for the difference in group means, we can use `resample()` and `do()` to generate a bootstrap distribution in one of two ways.

```
Temp.Boot2a <-
  do(1000) * diffmean(age ~ sex, data = resample(HELPrct))
Temp.Boot2b <-
  do(1000) * diffmean(age ~ sex, data = resample(HELPrct, groups = sex))
```

In the second example, the resampling happens within the sex groups so that the marginal counts for each sex remain fixed. This can be especially important if one of the groups is small, because otherwise some resamples might not include any observations of that group.

Note
It is useful to adopt a convention regarding the naming of randomization and bootstrap distributions. The names should reflect that data being used and whether the distribution is a bootstrap distribution or a randomization distribution. We typically use `.Rand` or `.Null` to indicate randomization distributions and `.Boot` to indicate bootstrap distributions.

```
favstats(age ~ sex, data = HELPrct)

     sex min Q1 median   Q3 max mean   sd   n missing
1 female  21 31     35 40.5  58 36.3 7.58 107       0
2   male  19 30     35 40.0  60 35.5 7.75 346       0
```

```
D <- diffmean( age ~ sex, data = HELPrct); D

diffmean
  -0.784
```

```
favstats(age ~ sex, data = resample(HELPrct))

     sex min Q1 median Q3 max mean   sd   n missing
1 female  21 30     34 39  58 35.3 7.80  98       0
2   male  19 30     35 39  58 35.3 7.59 355       0
```

```
favstats(age ~ sex, data = resample(HELPrct, groups = sex))
```

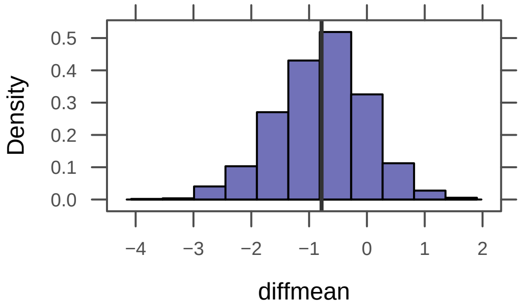
```
     sex min Q1 median Q3 max mean   sd   n missing
1 female  21 31     37 40  58 37.2 7.93 107       0
2   male  19 30     35 41  60 35.7 7.95 346       0
```

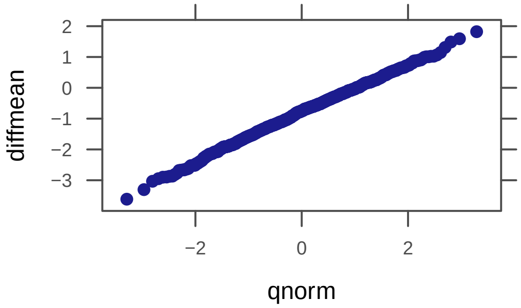
From here, the computation of confidence intervals proceeds as before.

```
histogram( ~ diffmean, data = Temp.Boot2b, v = D)
qqmath( ~ diffmean, data = Temp.Boot2b)
cdata( ~ diffmean, p = 0.95, data = Temp.Boot2b)
```

```
     low        hi central.p
  -2.444     0.702     0.950
```

Note
Visually inspecting the bootstrap distribution for skew and bias is an important step to make sure the percentile interval is not being applied in a situation where it may perform poorly.





Alternatively, we can compute a confidence interval based on a bootstrap estimate of the standard error.

```
SE <- sd( ~ diffmean, data = Temp.Boot2b); SE
```

```
[1] 0.789
```

```
D + c(-1,1) * 2 * SE
```

```
[1] -2.362  0.794
```

Either interval can be computed using `confint()`, if we prefer.

```
confint(Temp.Boot2b, method = c("percentile", "stderr"))
```

```
      name lower upper level     method estimate
1 diffmean -2.44 0.702  0.95 percentile   -0.784
2 diffmean -2.34 0.758  0.95     stderr   -0.784
  margin.of.error  df
1              NA  NA
2            1.55 452
```

5.3.4 Bootstrap distributions comparison

To illustrate the similarity among commands used to create bootstrap distributions, we present five examples that might appear in an introductory course.

```
 One.Proportion  <- do(1000) *     prop(   ~ x, data = resample(Data))
 Two.Proportions <- do(1000) * diffprop( y ~ x, data = resample(Data, groups = x))
 One.Mean        <- do(1000) *     mean(   ~ x, data = resample(Data))
 Two.Means       <- do(1000) * diffmean( y ~ x, data = resample(Data, groups = x))
 Correlation     <- do(1000) *      cor( y ~ x, data = resample(Data))
```

In the next section we discuss how to extend this to regression models.

5.4 Resampling for Regression

There are at least two ways we can consider creating a bootstrap distribution for a linear model. We can easily fit a linear model to a resampled data set. But in some situations this may have undesirable features. Influential observations, for example, will appear duplicated in some resamples and be missing entirely from other resamples.

Another option is to use “residual resampling". In residual resampling, the new data set has all of the predictor values from the original data set and a new response is created by adding to the fitted function a resampled residual.

Both methods are simple to implement; we either resample the data or resample the model itself.

```
mod <- lm( length ~ width + sex, data = KidsFeet)         # original model
do(1) * mod                                               # see how do() treats it

  Intercept width   sexG sigma r.squared    F numdf dendf
1      10.4   1.6 -0.133  1.04     0.413 12.7     2    36

do(2) * lm( length ~ width + sex, data = resample(KidsFeet))  # resampled data

  Intercept width   sexG sigma r.squared     F numdf dendf
1     11.39  1.48 -0.132 0.981     0.338  9.17     2    36
2      7.46  1.95 -0.338 1.005     0.485 16.92     2    36

do(2) * lm( length ~ width + sex, data = resample(mod))       # resampled residuals

  Intercept width    sexG sigma r.squared    F numdf dendf
1      5.84  2.13 -0.1113 0.849     0.642 32.3     2    36
2      9.20  1.74 -0.0222 0.979     0.465 15.7     2    36

do(2) * relm(mod)                 # abbreviated residual resampling

  Intercept width    sexG sigma r.squared     F numdf dendf
1      13.3  1.26 -0.0696 0.913     0.355  9.92     2    36
2       4.5  2.24  0.0278 1.093     0.534 20.59     2    36
```

From here it is straightforward to create a confidence interval for the slope (or intercept, or any coefficient) in a linear model.

```
Kids.Boot <- do(1000) * relm(mod)
cdata( ~ width, data = Kids.Boot, p = 0.95)

      low        hi central.p
    0.971     2.214     0.950

confint( Kids.Boot, parm = "width")

Warning:  confint:  Using df=Inf.

   name lower upper level method estimate margin.of.error
1 width 0.947  2.24  0.95 stderr     1.44           0.647
```

5.5 Which comes first: p-values or intervals?

This is a matter of some discussion among instructors and textbook authors. The two most recognizable introductory statistics books give different answers. One[7] introduces hypothesis testing first, the other[8] begins with bootstrap confidence intervals. These two books differ in several other ways as well. It remains to be seen whether best practices will emerge or whether some issues will remain a matter of personal preference. This is not unlike the older debate over whether one should begin with quantitative or categorical data – another way in which these two simulation-based books diverge.

5.6 Dealing with Monte Carlo Variability

Because randomization and bootstrap distributions involve a random component, p-values and confidence intervals computed from the same data will vary. For students (and graders), this can be disconcerting because there is no “right" answer.

The amount of Monte Carlo variability depends on the number of replicates used to create the randomization or bootstrap distribution. And students will need some guidance about how many replicates to use. It is important that they not use too few as this will introduce too much random noise into p-value and confidence interval calculations. But each replicate costs time, and the marginal gain for each additional replicate decreases as the number of replicates incresases. There is little reason to use millions of replicates (unless the goal is to estimate very small p-values). We generally use roughly 1000 for routine or preliminary work and increase this to 10,000 when we want to reduce the effects of Monte Carlo variability.

In a laboratory setting, it can be instructive to have students compare their p-values or confidence intervals using 1,000 and 10,000 replicates. Alternatively, the instructor can generate several p-values or confidence intervals to illustrate the same principle.

[7] Nathan Tintle, Beth Chance, George Cobb, Allan Rossman, Soma Roy, Todd Swanson, and Jill VanderStoep. *Introduction to Statistical Investigations*. Wiley Global Education, 2015

[8] Robin H Lock, Patti Frazer Lock, and Kari Lock Morgan. *Statistics: Unlocking the Power of Data*. Wiley Global Education, 2012

5.7 *Better Confidence Intervals*

The percentile and "t with bootstrap standard error" confidence intervals have been improved upon in a number of ways. In a first course, we generally do little more than mention this fact, and encourage students to inspect the shape of bootstrap distribution for indications of potential problems with the percentile method.

One improvement that can be explained to students in a course that combines simulation-based and formula-based approaches is the bootstrap-t interval. Rather than attempting to determine the best degrees of freedom for a Student's t-distribution, the bootstrap-t approximates the actual distribution of

$$t = \frac{\hat{\theta} - \theta}{SE}$$

using the boostrap distribution of

$$t^* = \frac{\hat{\theta}^* - \hat{\theta}}{SE^*} \,,$$

where $\hat{\theta}^*$ and SE^* are the estimate and estimated standard error computed from each bootstrap distribution. Implementing the bootstrap-t interval requires either an extra level of conceptual framework or much more calculation to determine the values of SE^*. If a standard error formula exists (e.g., $SE = s/\sqrt{n}$), this can be applied to each bootstrap sample along with the estimator. An alternative is to iterate the bootstrap procedure (resampling from each resample) to estimate SE^*. Since standard errors are easier to estimate than confidence intervals, fewer resamples are required (per resample) at the second level; nevertheless, the additional computational overhead is significant.

The `mosaic` package does not attempt to provide a general framework for the bootstrap-t or other "second-order accurate" boostrap methods. Packages such as `resample`[9] and `boot`[10] are more appropriate for situations where speed and accuracy are of utmost importance. But the bootstrap-t confidence interval can be computed using `confint()`, `do()` and `favstats()` in the case of estimating a single mean or the difference between two means.

[9] Tim Hesterberg. *resample: Resampling Functions*, 2015. R package version 0.4

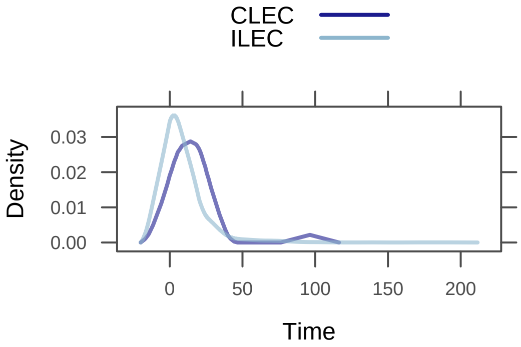
[10] Angelo Canty and Brian Ripley. *boot: Bootstrap R (S-Plus) Functions*, 2015. R package version 1.3-17; and A. C. Davison and D. V. Hinkley. *Bootstrap Methods and Their Applications*. Cambridge University Press, 1997

In the example below, we analyse a data set from the `resample` package. The `Verizon` data set contains repair times for customers in CLEC (competitive) and ILEC (incumbant) local exchange carrior.

```
# the resample package has name collisions with mosaic,
# so we only load the data, not the package
data(Verizon, package = "resample")
ILEC <- Verizon %>% filter(Group == "ILEC")
favstats( ~ Time, groups = Group, data = Verizon)
```

```
  Group min   Q1 median    Q3   max  mean   sd    n missing
1  CLEC   0 5.43  14.33 20.71  96.3 16.51 19.5   23       0
2  ILEC   0 0.73   3.59  7.08 191.6  8.41 14.7 1664       0
```

```
ashplot( ~ Time, groups = Group, data = Verizon,
         auto.key = TRUE, width = 20)
```



The skewed distributions of the repair times and unequal sample sizes highlight differences between the bootstrap-t and simpler methods.

```
BootT1 <-
   do(1000) * favstats(~ Time, data = resample(ILEC))
confint(BootT1, method = "boot")
```

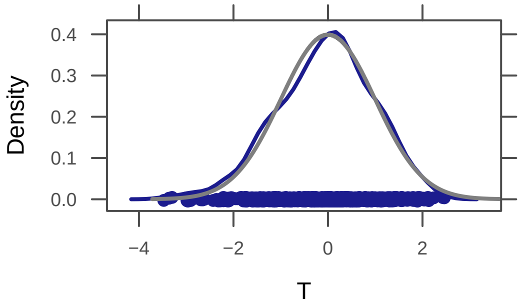
```
  name lower upper level      method estimate
1 mean  7.76  9.19  0.95 bootstrap-t     8.41
```

```
BootT2 <-
  do(1000) * favstats( ~ Time, groups = Group,
                       data = resample(Verizon, groups = Group))
confint(BootT2, method = "boot")
```

```
      name lower upper level      method estimate
1 diffmean -22.5 -2.57  0.95 bootstrap-t     -8.1
```

This can also be accomplished manually, although the computations are a bit involved for the 2-sample case. Here are the manual computations for the 1-sample case:

```
estimate <- mean( ~ Time, data = ILEC)
estimate

[1] 8.41

SE <- sd( ~ mean, data = BootT1); SE

[1] 0.361

BootT1a <-
  BootT1 %>%
  mutate( T = (mean - mean(mean)) / (sd/sqrt(n)))
q <- quantile(~ T, p = c(0.975, 0.025), data = BootT1a)
q

97.5%  2.5%
 1.81 -2.15

estimate - q * SE

97.5%  2.5%
 7.76  9.19

densityplot( ~ T, data = BootT1a)
plotDist("norm", add = TRUE, col="gray50")
```



For comparison, here are the intervals produced by `t.test()` and the percentile method.

```
confint(t.test( ~ Time, data = ILEC))

  mean of x lower upper level
1      8.41  7.71  9.12  0.95

BootT1b <-
  do(1000) * mean( ~ Time, data = resample(ILEC))
confint(BootT1b, method = "perc")

  name lower upper level     method estimate
1 mean   7.7   9.1  0.95 percentile     8.41


confint(t.test(Time ~ Group, data = Verizon))

  mean in group CLEC mean in group ILEC  lower upper level
1               16.5               8.41 -0.362  16.6  0.95

BootT2b <-
  do(1000) * diffmean(Time ~ Group, data = resample(Verizon, groups = Group))
confint(BootT2b, method = "perc")

      name lower upper level     method estimate
1 diffmean -16.6 -1.69  0.95 percentile     -8.1
```

In a situation like this, the intervals produced by t.test() are narrower, do the least to compensate for skew, undercover, and miss more often in one direction than in the other.

Even if these methods are not presented to students, it is good for instructors to be at least somewhat familiar with the issues involved and some of the methods that have been developed to handle them.[11]

5.8 *Simulating sampling distributions*

We conclude this chapter with one more use of sample(). If we treat a data frame as a population, sample() can be used to draw random samples of a specified size to illustrate the idea of a sampling distribution. We could use this to illustrate the sampling distribution of a sample mean, for example.

As an example, we will use the NHANES data. This data set has been adjusted to reflect the sampling weights used

[11] Tim C. Hesterberg. What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum. *The American Statistician*, 2015

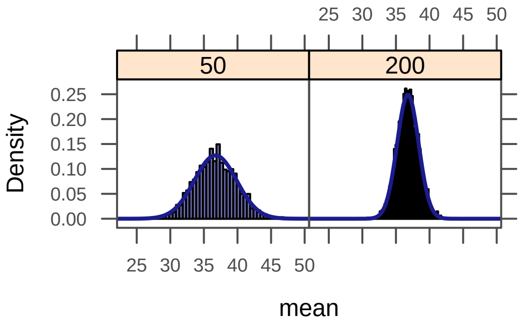
in the American National Health and Nutrition Examination surveys and is a reasonably good approximation to a simple random sample of size 10,000 from the US population. For the purpose of this example, we will treat this as the entire population and consider samples drawn from it, focusing (for the moment) on the `Age` variable.

```
require(NHANES)
mean( ~ Age, data = NHANES)          # population mean

[1] 36.7
```

We will consider samples of size 50 and size 200. This can be used to demonstrate the role of sample size in the sampling distribution.

```
mean( ~ Age, data = sample(NHANES, 50))  # mean of one sample

[1] 34.1

mean( ~ Age, data = sample(NHANES, 50))  # mean of another sample

[1] 38.4


# We use bind_rows() to combine two sampling distributions
# (with different sample sizes) into a single data frame to
# make graphical and numerical summaries easier.
SamplingDist <-
  bind_rows(
    do(2000) * c(mean = mean( ~ Age, data = sample(NHANES, 50)), n= 50),
    do(2000) * c(mean = mean( ~ Age, data = sample(NHANES, 200)), n= 200)
  )


mean( mean ~ n, data = SamplingDist)    # mean of sampling distribution

  50  200
36.7 36.8

sd( mean ~ n, data = SamplingDist)      # SE from sampling distribution

  50  200
3.14 1.60
```

```
sd( ~ Age, data = NHANES) / c("50" = sqrt(50), "200" = sqrt(200))  # SE from formula

  50  200
3.17 1.58

histogram( ~ mean | factor(n), data = SamplingDist,
           nint = 50, density = TRUE)
```



A similar approach can be used to create sampling distributions in other situations.

6
What Students Need to Know About R & How to Teach It

In Chapter 2, we give a brief orientation to the RStudio IDE and what happens in each of its tabs and panels. In Chapter 4, we show how to make use of a common template for graphical summaries, numerical summaries, and modeling. In this chapter we cover some additional things that are important for students to know about the R language.

More Info
Be sure to look at *A Student's Guide to R* as well. That little book contains a brief summary of all the commands needed to perform the statistical analyses typically seen in the first two statistics courses.

6.1 Two Questions

When we introduced the formula template in Chapter 4, we presented two important questions to ask before constructing an R command. These questions are useful in contexts beyond the formula template, and indeed for computer systems beyond R, so we repeat them here.

1. What do you want R to do?

   This will generally determine which R function to use.

2. What must R know to do that?

   This will determine the inputs to the function.

When your students preface their questions about R by telling you what they want R to do and what R needs to know to that, then you know they have internalized these two questions.

Teaching Tip
When students have difficulty accomplishing a task in R, make sure they can answer these questions before you show them what to do. If they cannot answer these questions, then the primary problem is not with R. If you do this consistently, eventually, you will find your students presenting their R questions to you by answering these two questions and then asking “So how do I get R to do that?" More likely, once they have answered these two questions, they will already know how to get R to do what they want – unless they are asking about functionality that you have not yet presented.

6.2 Four Things to Know About R

As is true for most computer languages, R has to be used on its terms. R does not learn the personality and style of its users. Getting along with R is much easier if you keep in mind (and remind your students about) a few key features of the R language.

1. R is case-sensitive

   If you mis-capitalize something in R it won't do what you want. Unfortunately, there is not a consistent convention about how capitalization should be used, so you just have to pay attention when encountering new functions and data sets.

2. Functions in R use the following syntax:

   ```
   functionname( argument1, argument2, ... )
   ```

   - The arguments are always *surrounded by (round) parentheses* and *separated by commas*.

     Some functions (like `data()`) have no required arguments, but you still need the parentheses.
   - If you type a function name without the parentheses, you will see the *code* for that function (this generally isn't what you want unless you are curious about how something is implemented).

3. TAB completion and arrows can improve typing speed and accuracy.

   If you begin a command and hit the TAB key, R and RStudio will show you a list of possible ways to complete the command. If you hit TAB after the opening parenthesis of a function, RStudio will display the list of arguments it expects.

   The up and down arrows can be used to retrieve past commands when working in the console.

4. If you see a + prompt, it means R is waiting for more input.

   Often this means that you have forgotten a closing parenthesis or made some other syntax error. If you have messed up and just want to get back to the normal prompt, press the escape key and start the command fresh.

TEACHING TIP
Some students will be slow to catch on to the importance of capitalization. So you may have to remind them several times early on.

TEACHING TIP
Introduce functions by emphasizing the questions *What do we want the computer to do?* and *What information does the computer need to compute this?* The answer to the first question determines the function to use. The answer to the second question determines what the arguments must be.

CAUTION!
Your students will sometimes find themselves in a syntactic hole from which they cannot dig out. Teach them about the ESC key early.

6.3 Installing and Using Packages

R is open source software. Its development is supported by a team of core developers and a large community of users. One way that users support R is by providing **packages** that contain data and functions for a wide variety of tasks. As an instructor, you will want to select a few packages that support the way you want to teach your course.

If you need to install a package, most likely it will be on CRAN, the Comprehensive R Archive Network. Before a package can be used, it must be **installed** (once per computer or account) and **loaded** (once per R session). Installing downloads the package software and prepares it for use by compiling (if necessary) and putting its components in the proper location for future use. Loading makes a previously installed package available for use in an R session.

For example, to use the mosaic package, we must first install it:

```
install.packages("mosaic") # fetch package from CRAN
```

Once the package has been installed it must be *loaded* to make it available in the current session or file using

```
library(mosaic)          # load the package before use
library(mosaicData)      # load data sets too
```

or

```
require(mosaic)           # alternative way to load
require(mosaicData)       # load data sets too
```

The Packages tab in RStudio makes installing and loading packages particularly easy and avoids the need for install.packages() for packages on CRAN, and makes loading packages into the console as easy as selecting a check box. The require() (or library()) function is still needed to load packages within RMarkdown, knitr/LaTeX, and script files.

If you are running on a machine where you don't have privileges to write to the default library location, you can

TEACHING TIP

If you set up an RStudio server, you can install all of the packages you want to use. You can even configure the server to autoload packages you use frequently. Students who use R on their desktop machines will need to know how to install and load these packages, however.

TEACHING TIP

The use of library() is more common in this situation, but we find that students remember the word require() better. For their purposes, the two are essentially the same. The biggest difference is how they respond when a package cannot be loaded (usually because it has not been installed). require() generates a warning message and returns a logical value that can be used when programming. library() generates an error when the package cannot be loaded.

MORE INFO

Even though the command is called library(), the thing loaded is a *package*, not a library.

CAUTION!

Remember that in RMarkdown and Rnw files, any packages you use must be loaded *within the file*.

install a personal copy of a package. If the location of your personal library is first in R_LIBS, this will probably happen automatically. If not, you can specify the location manually:

```
install.packages("mosaic", lib="~/R/library")
```

CRAN is not the only repository of R packages. Bioconductor is another large and popular repository, especially for biological applications, and increasingly authors are making packages available via github. For example, you can also install the mosaic package using

```
# if you haven't already installed devtools
install.packages("devtools")
require(devtools)
install_github("ProjectMOSAIC/mosaic")
```

Occasionally you might find a package of interest that is not available via a repository like CRAN or Bioconductor. Typically, if you find such a package, you will also find instructions on how to install it. If not, you can usually install directly from the zipped up package file.

```
# repos = NULL indicates to use a file, not a repository
install.packages('some-package.tar.gz', repos=NULL)
```

From this point on, we will assume that the mosaic package has been installed and loaded.

6.4 Getting Help

If something doesn't go quite right, or if you can't remember something, it's good to know where to turn for help. In addition to asking your friends and neighbors, you can use the R help system.

6.4.1 ?

To get help on a specific function or data set, simply precede its name with a ?:

```
?log       # help for the log function
```

```
?HELPrct  # help on a data set in the mosaic package
```

This will give you the documentation for the object you are interested in.

6.4.2 apropos()

If you don't know the exact name of a function, you can give part of the name and R will find all functions that match. Quotation marks are mandatory here.

```
apropos('tally')   # must include quotes. single or double.
```

```
[1] "statTally" "tally"     "tally"
```

More Info

Notice that tally appears twice. That is because there are two tally() functions, one in the mosaic package and one in the dplyr package. The find() function can be used to determine which package(s) a function belongs to. In this case, the mosaic package takes care of navigating among the two versions of tally(). In other cases, you may need to explicitly specify which package's function you want.

6.4.3 ?? and help.search()

If that fails, you can do a broader search using ?? or help.search(), which will find matches not only in the names of functions and data sets, but also in the documentation for them. Quotation marks are optional here.

6.4.4 Examples and Demos

Many functions and data sets in R include example code demonstrating typical uses. For example,

```
example(histogram)
```

will generate a number of example plots (and provide you with the commands used to create them). Examples such as this are intended to help you learn how specific R functions work. These examples also appear at the end of the documentation for functions and data sets.

Not all package authors are equally skilled at creating examples. Some of the examples are nonexistent or next to useless, others are excellent.

The mosaic package (and some other packages as well) also includes demos. Demos are bits of R code that can be executed using the demo() command with the name of the demo. To see how demos work, give this a try:

```
demo(lattice)
```

Demos are intended to illustrate a concept or a method and are independent of any particular function or data set.

You can get a list of available demos using

```
demo()                      # all demos
demo(package='mosaic')      # just demos from mosaic package
```

6.5 Data

6.5.1 Data Frames

Data sets are usually stored in a special structure called a **data frame**.

TEACHING TIP
Students who collect their own data, especially if they store it in Excel, are unlikely to put data into the correct format unless explicitly taught to do so.

Data frames have a 2-dimensional structure.

- Rows correspond to **observational units** (people, animals, plants, or other objects we are collecting data about).
- Columns correspond to **variables** (measurements collected on each observational unit).

TEACHING TIP
To help students keep variables and data frames straight, and to make it easier to remember the names, we have adopted the convention that data frames in the mosaicData package are capitalized and variables (usually) are not. This convention has worked well, and you may wish to adopt it for your data sets as well.

The Births78 data frame contains four variables measured for each day in 1978. There are several ways we can get some idea about what is in the Births78 data frame.

```
head(Births78)            # show the first few rows
```

```
        date births dayofyear  wday
1 1978-01-01   7701         1   Sun
2 1978-01-02   7527         2   Mon
3 1978-01-03   8825         3  Tues
4 1978-01-04   8859         4   Wed
5 1978-01-05   9043         5 Thurs
6 1978-01-06   9208         6   Fri
```

```
sample(Births78, 4)       # show 4 randomly selected rows
```

```
          date births dayofyear  wday orig.id
105 1978-04-15   7527       105   Sat     105
287 1978-10-14   8554       287   Sat     287
149 1978-05-29   7780       149   Mon     149
320 1978-11-16   9568       320 Thurs     320
```

```
summary(Births78)         # provide summary info about each variable
```

```
      date                births          dayofyear
 Min.   :1978-01-01   Min.   : 7135   Min.   :  1
 1st Qu.:1978-04-02   1st Qu.: 8554   1st Qu.: 92
 Median :1978-07-02   Median : 9218   Median :183
 Mean   :1978-07-02   Mean   : 9132   Mean   :183
 3rd Qu.:1978-10-01   3rd Qu.: 9705   3rd Qu.:274
 Max.   :1978-12-31   Max.   :10711   Max.   :365

    wday
 Sun  :53
 Mon  :52
 Tues :52
 Wed  :52
 Thurs:52
 Fri  :52
 Sat  :52
```

```
inspect(Births78)         # provide summary info about each variable
```

```
categorical variables:
  name   class levels   n missing                                  distribution
1 wday ordered      7 365       0 Sun (14.5%), Mon (14.2%), Tues (14.2%) ...

quantitative variables:
       name   class  min   Q1 median   Q3   max mean  sd   n missing
```

```
1    births integer 7135 8554   9218 9705 10711 9132 818 365       0
2 dayofyear integer    1   92    183  274    365  183 106 365       0

time variables:
  name   class      first       last min_diff max_diff   n missing
1 date POSIXct 1978-01-01 1978-12-31        1        1 365       0
```

```
str(Births78)             # show the structure of any R object
```

```
'data.frame': 365 obs. of  4 variables:
 $ date     : POSIXct, format: "1978-01-01" ...
 $ births   : int  7701 7527 8825 8859 9043 9208 8084 7611 9172 9089 ...
 $ dayofyear: int  1 2 3 4 5 6 7 8 9 10 ...
 $ wday     : Ord.factor w/ 7 levels "Sun"<"Mon"<"Tues"<..: 1 2 3 4 5 6 7 1 2 3 ...
```

The output from `str()` is also available in the Environment tab.

In interactive mode, you can also try

```
?Births78
```

to access the documentation for the data set. This is also available in the Help tab. Finally, the Environment tab provides a list of data in the global environment. Clicking on one of the data sets brings up the same data viewer as

```
View(Births78)
```

We can gain access to a single variable in a data frame using the $ operator or, alternatively, using the `with()` function.

```
dataframe$variable
with(dataframe, variable)
```

For example, either of

```
Births78$births
with(Births78, births)
```

will show the contents of the `births` variable in `Births78` data set.

Listing the entire set of values for a particular variable isn't very useful for a large data set. We would prefer to

As we will see, there are relatively few instances where one needs to use the $ operator.

compute numerical or graphical summaries. We'll do that shortly.

6.5.2 The Perils of attach()

The attach() function in R can be used to make objects within data frames accessible in R with fewer keystrokes, but we strongly discourage its use, as it often leads to name conflicts and other complications. The Google R Style Guide[1] echoes this advice, stating that

Caution!
Avoid the use of attach().

> *The possibilities for creating errors when using* attach() *are numerous. Avoid it.*

It is far better to directly access variables using the $ syntax or to use functions that allow you to avoid the $ operator.

6.5.3 Data in Packages

Teaching Tip
Start out using data in packages and show students how to import their own data once they understand how to work with data.

Data sets in R packages are the easiest to deal with. In section 6.5.4, we'll describe how to load your own data into R and RStudio, but we recommend starting with data in packages, and that is what we will do here, too. Once students know how to work with data and what data in R are supposed to look like, they will be better prepared to import their own data sets.

Many packages contain data sets. You can see a list of all data sets in all loaded packages using

```
data()
```

You can optionally choose to restrict the list to a single package:

```
data(package="mosaic")
```

Typically you can use data sets by simply typing their names. But if you have already used that name for something or need to refresh the data after making some changes you no longer want, you can explicitly load the data using the data() function with the name of the data set you want.

More Info
This depends on the package. Most package authors set up their packages with "lazy loading" of data. If they do not, then you need to use data() explicitly.

[1] http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html

```
data(Births78)
```

There is no visible effect of this command, but the Births78 data frame has now been reloaded from the mosaicData package and is ready for use. Anything you may have previously stored in a variable with this same name is replaced by the version of the data set stored with in the mosaicData package.

Caution!
If two packages include data sets with the same name, you may need to specify which package you want the data from with `data(Births78, package="mosaicData")`

6.5.4 Using Your Own Data

Teaching Tip
Start out using data from packages and focusing on what R can do with the data. Later, once students are familiar with R and understand the format required for data, teach students how to import their own data.

Eventually, students will want to move from using example data sets in R packages to using data they find or collect themselves. When this happens will depend on the type of students you have and the type of course you are teaching.

R provides the functions `read.csv()` (for comma separated values files), `read.table()` (for white space delimited files) and `load()` (for loading data in R's native format). The mosaic package includes a function called `read.file()` that uses slightly different default settings and infers whether it should use `read.csv()`, `read.table()`, or `load()` based on the file name.

Since most software packages can export to csv format, this has become a sort of *lingua franca* for moving data between packages. Data in excel, for example, can be exported as a csv file for subsequent reading in R. There is a danger in doing this, however, since some types of data don't export from Excel they way you might expect. A safer way to read excel files is to use the `read_excel()` function from the readxl package. The haven package includes utilities for reading data in several other formats that are exported from other statistics packages like SAS and Stata.

Some of these data ingesting functions accept a URL as well as a file name, which provides an easy way to distribute data via the Internet:

```
births <-
  read.table('http://www.calvin.edu/~rpruim/data/births.txt', header=TRUE)
head(births) # live births in the US each day of 1978.
```

```
    date births datenum dayofyear
1 1/1/78   7701    6575         1
2 1/2/78   7527    6576         2
3 1/3/78   8825    6577         3
4 1/4/78   8859    6578         4
5 1/5/78   9043    6579         5
6 1/6/78   9208    6580         6
```

We can omit the `header=TRUE` if we use `read.file()`

```
births <-
  read.file('http://www.calvin.edu/~rpruim/data/births.txt')
```

Reading data with read.table()

6.5.5 *Importing Data in* RStudio

The RStudio interface provides some GUI tools for loading data. If you are using the RStudio server, you will first need to upload the data to the server (in the Files tab), and then import the data into your R session (in the Environment tab).

If you are running the desktop version, the upload step is not needed.

Teaching Tip
Remind students that the 2-step process (upload, then import) works much like images in Facebook. First you upload them to Facebook, and once they are there you can include them in posts, etc.

6.5.6 *Working with Pretabulated Data*

Because categorical data is so easy to summarize in a table, often the frequency or contingency tables are given instead. You can enter these tables manually using a combination of `c()`, `rbind()` and `cbind()`:

Even if you use RStudio GUI for interactive work, you will want to know how to use functions like `read.csv()` for working in RMarkdown, or `knitr`/LaTeX files.

```
myrace <- c( NW=67, W=467 )  # c for combine or concatenate
myrace
```

```
 NW   W
 67 467
```

Teaching Tip
This is an important technique if you use a text book that presents pre-tabulated categorical data.

```
mycrosstable <- rbind(
  NW = c(clerical=15, const=3, manag=6,  manuf=11,
                 other=5,  prof=7, sales=3, service=17),
  W  = c(82,17,49,57,63,98,35,66)
                          )
mycrosstable
```

```
   clerical const manag manuf other prof sales service
NW       15     3     6    11     5    7     3      17
W        82    17    49    57    63   98    35      66
```

Replacing rbind() with cbind() will allow you to give the data column-wise instead.

This arrangement of the data would be sufficient for applying the Chi-squared test, but it is not in a format suitable for plotting with lattice. Our cross table is still missing a bit of information – the names of the variables being stored. We can add this information if we convert it to a table:

Teaching Tip
If plotting pre-tabulated categorical data is important, you probably want to provide your students with a wrapper function to simplify all this. We generally avoid this situation by provided the data in raw format or by presenting an analysing the data in tables without using graphical summaries.

```
class(mycrosstable)
```

```
[1] "matrix"
```

```
mycrosstable <- as.table(mycrosstable)
```

```
# mycrosstable now has dimnames, but they are unnamed
dimnames(mycrosstable)
```

```
[[1]]
[1] "NW" "W"

[[2]]
[1] "clerical" "const"    "manag"    "manuf"    "other"
[6] "prof"     "sales"    "service"
```

```
# let's add meaninful dimnames
names(dimnames(mycrosstable)) <- c('race', 'sector')
mycrosstable
```

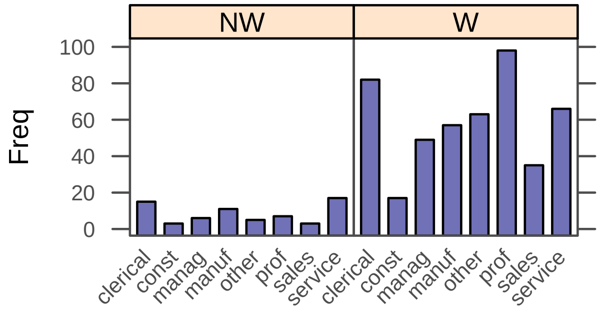
```
    sector
race clerical const manag manuf other prof sales service
  NW       15     3     6    11     5    7     3      17
  W        82    17    49    57    63   98    35      66
```

We can use `barchart()` instead of `bargraph()` to plot data already tabulated in this way, but first we need yet one more transformation.

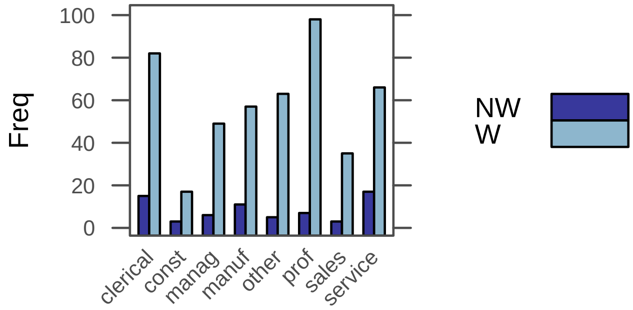
```
head(as.data.frame(mycrosstable))
```

```
  race   sector Freq
1   NW clerical   15
2    W clerical   82
3   NW    const    3
4    W    const   17
5   NW    manag    6
6    W    manag   49
```

```
barchart( Freq ~ sector | race,
          data=as.data.frame(mycrosstable),
          auto.key=list(space='right'),
          scales=list(x=list(rot=45))
                  )
```



```
barchart( Freq ~ sector, groups=race,
          data=as.data.frame(mycrosstable),
          auto.key=list(space='right'),
          scales=list(x=list(rot=45))
                  )
```



6.5.7 Developing Good Data Habits

However you teach students to collect and import their data, students will need to be trained to follow good data organization practices:

- Choose good variables names.
- Put variables names in the first row.
- Use each subsequent row for one observational unit.
- Give the resulting data frame a good name.

Some scientists may be disappointed that R data frames don't keep track of additional information, like the units in which the observations are recorded. This sort of information should be recorded, along with a description of the protocols used to collect the data, observations made during the data recording process, etc. This information should be maintained in a lab notebook or a **codebook**.

6.6 Review of R Commands

Here is a brief summary of the commands introduced in this chapter.

```
require(mosaic)                   # load the mosaic package
require(mosaicData)               # load the mosaic data sets
answer <- 42                      # store the number 42 in a variable named answer
log(123); log10(123); sqrt(123)   # some standard numerical functions
x <- c(1,2,3)                     # make a vector containing 1, 2, 3 (in that order)

data(iris)                        # (re)load the iris data set
names(iris)                       # see the names of the variables in the iris data
head(iris)                        # first few rows of the iris data set
sample(iris, 3)                   # 3 randomly selected rows of the iris data set
inspect(iris)                     # summarize each variable in the iris data set
summary(iris)                     # summarize each variable in the iris data set
str(iris)                         # show the structure of the iris data set

mydata <- read.table("file.txt")  # read data from a text file
mydata <- read.csv("file.csv")    # read data from a csv file
mydata <- read.file("file.txt")   # read data from a text or csv file
require(readxl)
mydata <- read_excel("file.xlsx") # read data from an Excel file
```

6.7 Exercises

6.1 The table below is from a study of nighttime lighting in infancy and eyesight (later in life).

	no myopia	myopia	high myopia
darkness	155	15	2
nightlight	153	72	7
full light	34	36	3

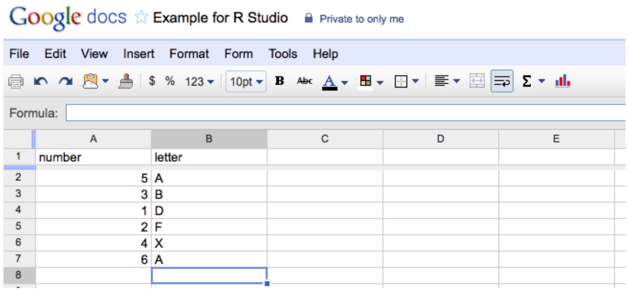
a) Recreate the table in R.

b) What percent of the subjects slept with a nightlight as infants?

There are several ways to do this. You could use R as a calculator to do the arithmetic. You can save some typing if you use the function `tally()`. See `?tally` for documentation.

c) Create a graphical representation of the data. What does this plot reveal?

6.2 Enter the following small data set in an Excel or Google spreadsheet and import the data into RStudio.



	A	B	C	D	E
1	number	letter			
2	5	A			
3	3	B			
4	1	D			
5	2	F			
6	4	X			
7	6	A			
8					

7
What Instructors Need to Know about R

We recommend keeping the amount of R that students need to learn to a minimum, and choosing functions that support a formula interface whenever possible to keep the required functions syntactically similar. But there are some additional things that instructors (and some students) should know about R. We outline some of these things in this chapter.

You may find that some of these things are useful for your students to know as well. That will depend on the goals for your course and the abilities of your students. In higher level courses, much of the material in this chapter is also appropriate for students.

7.1 Some Workflow Suggestions

Our workflow advice can be summarized in one short sentence:

Think like a programmer.

It doesn't take sophisticated programming skills to be good at using R. In fact, most uses of R for teaching statistics can be done working one step at a time, where each line of code does one complete and useful task. After inspecting the output (and perhaps saving it for further computation later), one can proceed to the next operation.

We don't really think of our classroom use of R as programming since we use R in a mostly declarative rather than algorithmic way.

Nevertheless, we can borrow from the collective wisdom of the programming community and adopt some practices that will make our experience more pleasurable, more efficient, and less error-prone.

- Store your code in a file.

  It can be tempting to do everything in the console. But the console is ephemeral. It is better to get into the habit of storing code in files. Get in the habit (and get your students in the habit) of working with R scripts and especially RMarkdown files.

You can execute all the code in an R script file using

```
source("file.R")
```

RStudio has additional options for executing some or all lines in a file. See the buttons in the tab for any R script, RMarkdown or Rnw file. (You can create a new file in the main File menu.)

If you work at the console's interactive prompt and later wish you had been putting your commands into a file, you can save your past commands with

```
savehistory("someRCommandsIalmostLost.R")
```

In RStudio, you can selectively copy portions of your history to a script file (or the console) using the History tab.

More Info
R can be used to create executable scripts. Option parsing and handling is supported with the optparse package.

- Use meaningful names.

  Rarely should objects be named with a single letter.

  Adopt a personal convention regarding case of letters. This will mean you have one less thing to remember when trying to recall the name of an object. For example, in the mosaicData package, all data frames begin with a capital letter. Most variables begin with a lower case letter (a few exceptions are made for some variables with names that are well-known in their capitalized form).

- Adopt reusable idioms.

  Computer programmers refer to the little patterns that recur throughout their code as idioms. For example, here is a "compute, save, display" idiom.

```
# compute, save, display idiom
footModel <- lm(length ~ width, data=KidsFeet); footModel


Call:
lm(formula = length ~ width, data = KidsFeet)

Coefficients:
(Intercept)        width
       9.82         1.66
```

```
# alternative that reflects the order of operations
lm(length ~ width, data=KidsFeet) -> footModel; footModel


Call:
lm(formula = length ~ width, data = KidsFeet)

Coefficients:
(Intercept)        width
       9.82         1.66
```

Often there are multiple ways to do the same thing in R, but if you adopt good programming idioms, it will be clearer to both you and your students what you are doing.

- Write reusable functions.

  Learning to write your own functions (see Section 7.7) will greatly increase your efficiency and also help you understand better how R works. This, in turn, will help you debug your students error messages. (More on error messages in 7.10.) It also makes it possible for you to simplify tasks you want your students to be able to do in R. That is how the `mosaic` package originated – as a collection of tools we had assembled over time to make teaching and learning easier.
- Comment your code.

  It's amazing what you can forget. The comment character in R is #. If you are working in RMarkdown or Rnw files, you can also include nicely formatted text to describe what you are doing and why.

7.2 *Primary* R *Data Structures*

Everything in R is an object of a particular kind and understanding the kinds of objects R is using demystifies many of the messages R produces and unexpected behavior when commands do not work the way you (or your students) were expecting. We won't attempt to give a comprehensive description of R's object taxonomy here, but will instead focus on a few important features and examples.

7.2.1 Objects and Classes

In R, data are stored in objects. Each **object** has a *name*, *contents*, and a *class*. The class of an object tells what kind of a thing it is. The class of an object can be queried using `class()`

More Info
Many objects also have *attributes* which contain additional information about the object, but unless you are doing programming with these objects, you don't need to worry much about them.

```
class(KidsFeet)

[1] "data.frame"

class(KidsFeet$birthmonth)

[1] "integer"

class(KidsFeet$length)

[1] "numeric"

class(KidsFeet$sex)

[1] "factor"

str(KidsFeet)                # show the class for each variable

'data.frame': 39 obs. of  8 variables:
 $ name      : Factor w/ 36 levels "Abby","Alisha",..: 10 24 36 20 23 34 13 4 14 8 ...
 $ birthmonth: int  5 10 12 1 2 3 2 6 5 9 ...
 $ birthyear : int  88 87 87 88 88 88 88 88 88 88 ...
 $ length    : num  24.4 25.4 24.5 25.2 25.1 25.7 26.1 23 23.6 22.9 ...
 $ width     : num  8.4 8.8 9.7 9.8 8.9 9.7 9.6 8.8 9.3 8.8 ...
 $ sex       : Factor w/ 2 levels "B","G": 1 1 1 1 1 1 1 2 2 1 ...
 $ biggerfoot: Factor w/ 2 levels "L","R": 1 1 2 1 1 2 1 1 2 2 ...
 $ domhand   : Factor w/ 2 levels "L","R": 2 1 2 2 2 2 2 2 2 1 ...
```

From this we see that `KidsFeet` is a data frame and that the variables are of different types (integer, numeric, and factor). These are the kinds of variables you are most likely to encounter, although you may also see variables that are logical (true or false) or character (text) as well.

Factors are the most common way for categorical data to be stored in R, but sometimes the character class is better. The class of an object determines what things can be done with it and how it appears when printed, plotted, or displayed in the console.

More Info
One difference between a factor and a character is that a factor knows the possible values, even if some them do not occur. Sometimes this is an advantage (tallying empty cells in a table) and sometimes it is a disadvantage (when factors are used as unique identifiers).

7.2.2 Containers

The situation is actually a little bit more complicated. The `birthmonth` variable in `KidsFeet` is not a single integer but a collection of integers. So we can think of `birthmonth` as a kind of container holding a number of integers. There is more than one kind of container in R. The containers used for variables in a data frame are called **vectors**. The items in a vector are ordered (starting with 1) and must all be of the same type.

More Info
Even when we only have a single integer, R will treat it like a container of integers with only one integer in it.

Vectors can be created using the `c()` function:

```
c(2, 3, 5, 7)

[1] 2 3 5 7

c("Abe", "Betty", "Chan")

[1] "Abe"   "Betty" "Chan"

c(1.2, 3.2, 4.5)

[1] 1.2 3.2 4.5
```

Caution!
When reading data created in other software (like Excel) or stored in CSV files, it is important to know how missing data were indicated, otherwise, the code for missing data may be interpreted as a character, causing all the other items in that column to be converted to character values as well, and losing the important information that some of the data were missing.

If you attempt to put different types of objects into a vector, R will attempt to convert them all to the same type of object. If it is unable to do so, it will generate an error.

```
x <- c(1, 1.1, 1.2); x    # convert integer to numeric

[1] 1.0 1.1 1.2

class(x)

[1] "numeric"

y <- c(TRUE, FALSE, 0, 1, 2); y     # logicals converted to numeric

[1] 1 0 0 1 2

class(y)

[1] "numeric"

z <- c(1, TRUE, 1.2, "vector"); z   # all converted to character

[1] "1"      "TRUE"   "1.2"    "vector"

class(z)

[1] "character"
```

Factors can be created by wrapping a vector with `factor()`:

```
w <- factor(x); w
```

```
[1] 1   1.1 1.2
Levels: 1 1.1 1.2
```

```
class(w)
```

```
[1] "factor"
```

Notice how factors display the **levels** (possible values) as well as the values themselves. When categorical data are coded as integers, it is important to remember to convert them to factors in this way for certain statistical procedures and some plots.

Digging Deeper
A factor can be ordered or unordered (which can affect how statistics tests are performed but otherwise does not matter much). The default is for factors to be unordered. Whether the factors are ordered or unordered, the levels will appear in a fixed order – alphabetical by default. The distinction between ordered and unordered factors has to do with whether this order is meaningful or arbitrary.

Patterned integer or numeric vectors can be created using the : operator or the `seq()` function.

```
 1:10
```

```
  [1]  1  2  3  4  5  6  7  8  9 10
```

```
 seq(1, 10, by=0.5)
```

```
  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0
 [12]  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0
```

Individual items in a vector can be accessed or assigned using the square bracket operator:

```
w[1]
```

```
[1] 1
Levels: 1 1.1 1.2
```

```
x[2]
```

```
[1] 1.1
```

```
y[3]
```

```
[1] 0
```

Missing values are coded as `NA` (not available). Asking for an entry “off the end” of a vector returns `NA`. Assigning a value "off the end" of a vector results in the vector being lengthened so that the new value can be stored in the appropriate location.

```
z[5]     # this is not an error, but returns NA (missing)

[1] NA

q <- 1:5
q

[1] 1 2 3 4 5

q[10] <- 10   # elements 6 thru 9 will be filled with NA
q

 [1]  1  2  3  4  5 NA NA NA NA 10
```

R also provides some more unusual (but very useful) features for accessing elements in a vector.

```
letters                       # alphabet

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
[15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

More Info
`letters` is a built-in character vector containing the lower case letters. LETTERS contains capitals.

```
x <- letters[1:10]; x         # first 10 letters

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

x[2:4]                        # select items 2 through 4

[1] "b" "c" "d"

x[2:4] <- c("X","Y","Z"); x   # change items 2 through 4

 [1] "a" "X" "Y" "Z" "e" "f" "g" "h" "i" "j"

y <- (1:10)^2; y              # first 10 squares

 [1]   1   4   9  16  25  36  49  64  81 100

y [ y > 20 ]                  # select the items greater than 20

[1]  25  36  49  64  81 100
```

The last item deserves a bit of comment. The expression inside the brackets evaluates to a vector of logical values.

```
y > 20
```

```
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

The logical values are then used to select (true) or deselect (false) the items in the vector, producing a new (and potentially shorter) vector. If the number of logical supplied is less than the length of the vector, the values are **recycled** (repeated).

```
y[ c(TRUE,FALSE) ]          # every other
```

```
[1]  1  9 25 49 81
```

```
y[ c(TRUE,FALSE,FALSE) ]    # every third
```

```
[1]   1  16  49 100
```

A **matrix** is a 2-dimensional table of values that all have the same type. As with vectors, all of the items in a matrix must be of the same type. But matrices are two-dimensional – each item is located in a row and column. An **array** is a multi-dimensional version of a matrix. Matrices and arrays are important containers for statistical work, but less likely to be encountered by beginners.

```
M <- matrix(1:15, nrow=3); M    # a 3 x 5 matrix
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
```

The dimensions of an array, matrix or data frame can be obtained using `dim()` or `nrow()` and `ncol()`.

```
dim(M)
```

```
[1] 3 5
```

```
dim(KidsFeet)
```

```
[1] 39  8
```

```
nrow(KidsFeet)
```

```
[1] 39
```

```
ncol(KidsFeet)
```

```
[1] 8
```

Another commonly used container in R is a list. We have already seen a few examples of lists used as arguments to `lattice` plotting functions. Lists are also ordered, but the items in a list can be objects of any type, and they need not all be the same type. Behind the scenes, a data frame is a list of vectors with the restriction that each vector must have the same **length** (contain the same number of items).

More Info

In official R parlance, the distinction we make between vectors and lists is really the distinction between *atomic* vectors and lists (which are also called *generic* vectors). In fact, they must all be of the same *atomic* type. Atomic vectors are are the basic building blocks for R. It is not possible to store more complicated objects (like data frames) in a vector, but they can be stored in a list.

Lists can be created using the `list()` function.

```
l <- list(1, "two", 3.2, list(1, 2)); l
```

```
[[1]]
[1] 1

[[2]]
[1] "two"

[[3]]
[1] 3.2

[[4]]
[[4]][[1]]
[1] 1

[[4]][[2]]
[1] 2
```

```
length(l)          # Note: l has 4 elements, not 5
```

```
[1] 4
```

Items in a list can be accessed with the double square bracket (`[[ ]]`).

```
l[[1]]
```

```
[1] 1
```

Using a single square bracket (`[ ]`) instead returns a sublist rather than an element. So `l[[1]]` is a vector, but `l[1]` is a list containing a vector.

```
l[1]

[[1]]
[1] 1
```

Both vectors and lists can be named. The names can be created when the vector or list is created or they can be added later. Elements of vectors and lists can be accessed by name as well as by position.

```
x <- c(one=1, two=2, three=3); x

  one   two three 
    1     2     3 

y <- list(a=1, b=2, c=3); y

$a
[1] 1

$b
[1] 2

$c
[1] 3

x["one"]

one 
  1 

y[["a"]]   # retrieve items from a list with [[ ]]

[1] 1

names(x)

[1] "one"   "two"   "three"

names(x) <- c("A", "B", "C"); x

A B C 
1 2 3 
```

The access operators – `[ ]` and `[[ ]]` for lists – are actually *functions* in R. This has some important consequences:

- Accessing elements in a vector is slower than in a language like C/C++ where access is done by pointer arithmetic.

- These functions also have named arguments, so you can see code like the following

```
M
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
```

```
M[5]
```

```
[1] 5
```

```
M[,2]                   # this is 1-d (a vector)
```

```
[1] 4 5 6
```

```
M[,2, drop=FALSE]       # this is 2-d (still a matrix)
```

```
     [,1]
[1,]    4
[2,]    5
[3,]    6
```

Data frames can be constructed by supplying `data.frame()` with the variables (as vectors):

```
ddd <- data.frame(number=1:5, letter=letters[1:5])
```

7.2.3 Vectorized functions

Vectors are so important in R that they deserve some additional discussion. Many R functions and operations are “vectorized” and can be applied not just to an individual value but to an entire vector, in which case they are applied componentwise and return a vector of transformed values. Most of the commonly used functions from mathematics are available and work this way.

```
x <- 1:5; y <- seq(10, 60, by=10)
x

[1] 1 2 3 4 5

y

[1] 10 20 30 40 50 60

y + 1                     # add 1 to each element

[1] 11 21 31 41 51 61

x * 10                    # multiply each element by 10

[1] 10 20 30 40 50

x < 3                     # check whether each is less than 3

[1]  TRUE  TRUE FALSE FALSE FALSE

x^2                       # square each element

[1]  1  4  9 16 25

sqrt(x)                   # square root of each element

[1] 1.00 1.41 1.73 2.00 2.24

log(x)                    # natural log

[1] 0.000 0.693 1.099 1.386 1.609

log10(x)                  # base 10 log

[1] 0.000 0.301 0.477 0.602 0.699
```

Vectors can be combined into a matrix using `rbind()` or `cbind()`. This can facilitate side-by-side comparisons.

```
# compare round() and signif() by binding row-wise into a matrix
z <- rnorm(5); z

[1] -0.5605 -0.2302  1.5587  0.0705  0.1293

rbind(round(z, digits=3), signif(z, digits=3))

      [,1]  [,2] [,3]   [,4]  [,5]
[1,] -0.56 -0.23 1.56 0.0710 0.129
[2,] -0.56 -0.23 1.56 0.0705 0.129
```

7.2.4 *Functions that act on vectors as vectors*

Other functions, including many statistical functions, are designed to compute a single number (technically, a vector of length 1) from an entire vector.

```
z <- rnorm(100)
# basic statistical functions; notice the use of names
c(mean=mean(z), sd=sd(z), var=var(z), median=median(z))
```

```
   mean      sd     var  median
 0.0607  0.9089  0.8260 -0.0114
```

```
range(z)                    # range returns a vector of length 2
```

```
[1] -2.31  2.19
```

```
x <- 1:10
c(sum=sum(x), prod=prod(x))   # sums and products
```

```
    sum    prod
     55 3628800
```

Still other functions return vectors that are derived from the original vector, but not as a componentwise transformation.

```
z <- rnorm(5); z

[1] -0.045 -0.785 -1.668 -0.380  0.919
```

```
sort(z); rank(z); order(z)

[1] -1.668 -0.785 -0.380 -0.045  0.919
[1] 4 2 1 3 5
[1] 3 2 4 1 5
```

```
x <- 1:10
rev(x)           # reverse x

 [1] 10  9  8  7  6  5  4  3  2  1

diff(x)          # pairwise differences

[1] 1 1 1 1 1 1 1 1 1

ediff(x)         # pairwise differences w/out changing length

 [1] NA  1  1  1  1  1  1  1  1  1

cumsum(x)        # cumulative sum

 [1]  1  3  6 10 15 21 28 36 45 55

cumprod(x)       # cumulative product

 [1]       1       2       6      24     120     720    5040
 [8]   40320  362880 3628800
```

Whether a function is vectorized or treats a vector as a unit depends on its implementation. Usually, things are implemented the way you would expect. Occasionally you may discover a function that you wish were vectorized and is not. When writing your own functions, give some thought to whether they should be vectorized, and test them with vectors of length greater than 1 to make sure you get the intended behavior.

More Info

The `Vectorize()` function is a useful tool for converting a non-vectorized function into a vectorized function.

The operations listed below can be helpful when writing your own functions.

`cumsum()`
`cumprod()`
`cummin()`
`cummax()`
	Returns vector of cumulative sums, products, minima, or maxima.

`pmin(x,y,...)`
`pmax(x,y,...)`
	Returns vector of parallel minima or maxima where *i*th element is max or min of `x[i]`, `y[i]`, ...

`which(x)`	Returns a vector of indices of elements of `x` that are true. Typical use: `which(y > 5)` returns the indices where elements of `y` are larger than 5.
`any(x)`	Returns a `logical` indicating whether any elements of `x` are true. Typical use: `if (any(y > 5)) { ...}`.
`na.omit(x)`	Returns a vector with missing values removed.
`unique(x)`	Returns a vector with repeated values removed.
`table(x)`	Returns a table of counts of the number of occurrences of each value in `x`. The table is similar to a vector with names indicating the values, but it is not a vector.

`paste(x,y,...,`
`sep=" ")`
	Pastes `x` and `y` together componentwise (as strings) with `sep` between elements. Recycling applies.

7.3 *Working with Data*

In Section 6.5 we discussed using data in R packages, and in Section 6.5.4 we discussed methods for bringing your own data into R. In both of these scenarios, we have assumed that the data had been entered and cleaned in some other software and focussed primarily on data import. In this section we discuss ways to create and manipulate data within R. But first we discuss a few more details regarding importing data.

Even if you primarily use the RStudio interface to import data, it is good to know about the command line methods since these are required to import data into scripts, RMarkdown, and `knitr`/LaTeX files.

7.3.1 *Finer control over data import*

The `na.strings` argument can be used to specify codes for missing values. Setting `na.strings` as in the following for reading csv files that might have been produced by systems such as SAS.

```
someData <- read.csv('file.csv',
  na.strings=c('NA','','.','-','na'))
```

SAS uses a period (.) to code missing data and some csv exporters use '-'. If the above definition for na.strings, or something like it, R will treat missing-data markers as string data, instead of NA. This forces the entire variable to be of character type even if it's otherwise purely numeric.

More Info
The read.file() function in the mosaic package uses this as its default for na.strings.

By default, R will recode character data as a factor. If you prefer to leave such variables in character format, you can use

```
someData <- read.file('file.csv',
  stringsAsFactors=FALSE)
```

More Info
This works with read.csv() and read.table() as well.

Reading data with read.csv()

Even finer control can be obtained by manually setting the class (type) used for each column in the file. In addition, this speeds up the reading of the file. For a csv file with four columns, we can declare them to be of class integer, numeric, character, and factor with the following command.

```
someData <- read.file('file.csv',
  na.strings=c('NA','','.','-','na'),
  colClasses=c('integer','numeric','character','factor'))
```

Reading data with read.csv()

7.3.2 Manually entering data

We have already seen that the c() function can be used to combine elements into a single vector.

```
x <- c(1, 1, 2, 3, 5, 8, 13); x
```

```
[1]  1  1  2  3  5  8 13
```

The scan() function can speed up data entry in the console by allowing you to avoid the commas. Individual values are separated by white space or new lines. A blank line is used to signal the end of the data. By default, scan() is expecting numeric data, but it is possible to tell scan() to expect something else, like **character** data (i.e., text). There are other options for data types, but numerical and text data handle the most important cases. See ?scan for more information and examples.

Caution!
When using scan() be sure to remember to save your data somewhere. Otherwise you will have to type it again.

7.3.3 *Simulating samples from distributions*

R has functions that make it simple to sample from a wide range of distributions. Each of these functions begins with the letter 'r' (for random) followed by the name of the distribution (often abbreviated somewhat). The arguments to the function specify the size of the sample desired and any parameter values required for the distribution. For example, to simulate selecting a sample of size 12 from a normal population with mean 100 and standard deviation 10, use

```
rnorm(12, mean=100, sd=10)
```

```
 [1]  94.2 106.1  83.8  99.4 105.2 103.0 101.1  93.6  91.5
[10]  89.8 101.2  90.5
```

Functions for sampling from other distributions include rbinom(), rchisq(), rt(), rf(), rhyper(), etc.

It is also easy to sample (with or without replacement) from existing data using sample() and resample().

```
x <- 1:10
# random sample of size 5 from x (no replacement)
sample(x, size=5)
```

```
[1]  4  7 10  9  6
```

```
# a different random sample of size 5 from x (no replacement)
sample(x, size=5)
```

```
[1]  8  3  2  5 10
```

```
# random sample of size 5 from x (with replacement)
resample(x, size=5)
```

```
[1] 6 8 2 5 5
```

Using resample() makes it easy to simulate small discrete distributions. For example, to simulate rolling 20 dice, we could use

```
resample(1:6, size=20)

 [1] 6 6 6 5 6 4 4 3 3 1 4 6 1 1 1 5 5 6 3 1
```

For working with cards, the mosaicData package provides a vector named Cards and deal() as an alternative name for sample().

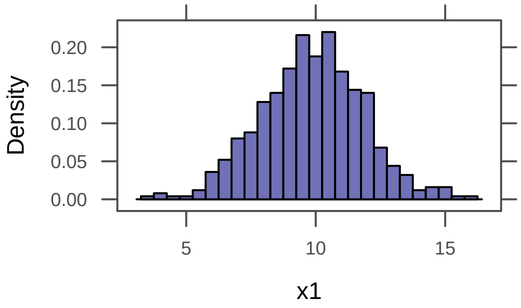
```
deal(Cards, 5)     # poker hand

[1] "9H" "AH" "8C" "8D" "QC"

deal(Cards, 13)    # bridge, anyone?

 [1] "5C" "9D" "AS" "KC" "4C" "7H" "2D" "6C" "QS" "KH" "9S"
[12] "9H" "2S"
```

If you want to sort the hands nicely, you can create a factor from Cards first:

```
 hand <- deal(factor(Cards, levels=Cards), 13)
 sort(hand)       # sorted by suit, then by denomination

  [1] 2C  7C  8C  7D  8D  10D 4H  9H  QH  AH  2S  10S AS 
 52 Levels: 2C 3C 4C 5C 6C 7C 8C 9C 10C JC QC KC AC ... AS
```

Example 7.1. For teaching purposes it is sometimes nice to create a histogram that has the approximate shape of some distribution. One way to do this is to randomly sample from the desired distribution and make a histogram of the resulting sample.

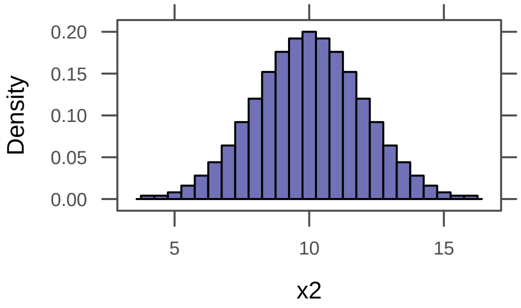
```
x1 <- rnorm(500, mean=10, sd=2)
histogram( ~ x1, width=.5)
```



This works, but the resulting plot has a fair amount of noise.

The ppoints() function returns evenly spaced probabilities and allows us to obtain theoretical quantiles of the normal distribution instead. The resulting plot now illustrates the idealized sample from a normal distribution.

```
x2 <- qnorm(ppoints(500), mean=10, sd=2)
histogram( ~ x2, width=.5)
```



This is not what real data will look like (even if it comes from a normal population), but it can be better for illustrative purposes to remove the noise. ◇

7.3.4 *Saving Data*

write.table() and write.csv() can be used to save data from R into delimited flat files.

```
ddd <- data.frame(number=1:5, letter=letters[1:5])
write.table(ddd, "ddd.txt")
write.csv(ddd, "ddd.csv")
```

Data can also be saved in native R format. Saving data sets (and other R objects) using save() has some advantages over other file formats:

- Complete information about the objects is saved, including attributes.
- Data saved this way takes less space and loads much more quickly.
- Multiple objects can be saved to and loaded from a single file.

The downside is that these files are only readable in R.

```
abc <- "abc"
ddd <- data.frame(number=1:5, letter=letters[1:5])
# save both objects in a single file
save(ddd, abc, file="ddd.rda")
# load them both
load("ddd.rda")
```

More Info
If you want to save an R object but not its name, you can use saveRDS() and choose its name when you read it with readRDS().

For more on importing and exporting data, especially from other formats, see the *R Data Import/Export* manual available on CRAN.

7.4 *Manipulating Data Frames with* dplyr

There are several ways to manipulate data frames in R. The approach illustrated here relies heavily on the functions in the dplyr package. This package is loaded when the mosaic package is loaded. The dplyr package defines five primary operations on a data frame

1. mutate() – add or change variables
2. select() – choose a subset of columns
3. filter() – choose a subset of rows
4. summarise() – reduce the entire data frame to a summary row
5. arrange() – reorder the rows

These become especially powerful when combined with a sixth command, group_by().

6. group_by() – split the data frame into multiple subsets

Additional functions (inner_join() and left_join() can be used to combine data from multiple data frames.

7.4.1 Adding new variables to a data frame

The mutate() function can be used to add or modify variables in a data frame.

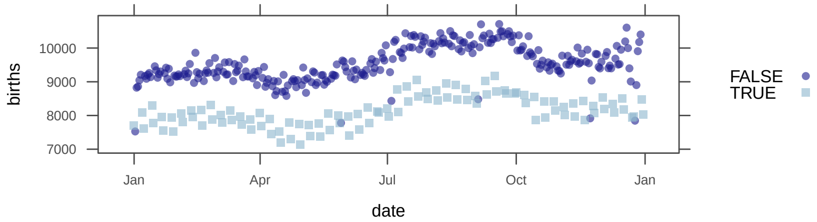
Here we show how to modify the Births78 data frame so that it contains a new variable weekend that distinguishes between weekdays and weekends.

NOTE

mutate() access to the other variables in the data frame, including any created earlier in the same mutate() command.

```
data(Births78)
weekdays <- c("Sun", "Mon", "Tue", "Wed", "Thr", "Fri", "Sat")
Births <-
  Births78 %>%
  mutate(weekend = wday %in% c("Sat", "Sun"))

head(Births, 3)

        date births dayofyear wday weekend
1 1978-01-01   7701         1  Sun    TRUE
2 1978-01-02   7527         2  Mon   FALSE
3 1978-01-03   8825         3 Tues   FALSE
```

```
xyplot(births ~ date, Births, groups=weekend, auto.key=list(space='right'))
```



Number of US births in 1978 colored by day of week.

The CPS85 data frame contains data from a Current Population Survey (current in 1985, that is). Two of the variables in this data frame are age and educ. We can estimate the number of years a worker has been in the workforce if we assume they have been in the workforce since completing their education and that their age at graduation is 6 more than the number of years of education obtained.

```
CPS85 <- mutate(CPS85, workforce.years = age - 6 - educ)
favstats( ~ workforce.years, data=CPS85)
```

```
 min Q1 median Q3 max mean   sd   n missing
  -4  8     15 26  55 17.8 12.4 534       0
```

In fact this is what was done for all but one of the cases to create the `exper` variable that is already in the `CPS85` data.

```
tally( ~ (exper - workforce.years), data=CPS85)
```

```

  0   4
533   1
```

With categorical variables, sometimes we want to modify the coding scheme.

```
HELP2 <- mutate(HELPrct,
  newsex = factor(female, labels=c('M','F')))
```

It's a good idea to do some sort of sanity check to make sure that the recoding worked the way you intended

```
tally( ~ newsex + female, data=HELP2)
```

```
      female
newsex   0   1
     M 346   0
     F   0 107
```

The `derivedFactor()` function can simplify creating factors based on some logical tests.

```
HELP3 <- mutate(HELPrct,
  risklevel = derivedFactor(
    low = sexrisk < 5,
  medium = sexrisk < 10,
        high = sexrisk >=10,
         .method = "first"     # use first rule that applies
         )
 )
head(HELP3, 4)
```

```
  age anysubstatus anysub cesd d1 daysanysub dayslink drugrisk e2b female    sex g1b
1  37            1    yes   49  3        177      225        0  NA      0   male yes
2  37            1    yes   30 22          2       NA        0  NA      0   male yes
3  26            1    yes   39  0          3      365       20  NA      0   male  no
4  39            1    yes   15  2        189      343        0   1      1 female  no
  homeless i1 i2 id indtot linkstatus link   mcs  pcs pss_fr racegrp satreat sexrisk
1   housed 13 26  1     39          1  yes 25.11 58.4      0   black      no       4
2 homeless 56 62  2     43         NA <NA> 26.67 36.0      1   white      no       7
3   housed  0  0  3     41          0   no  6.76 74.8     13   black      no       2
4   housed  5  5  4     28          0   no 43.97 61.9     11   white     yes       4
  substance treat risklevel
1   cocaine   yes       low
2   alcohol   yes    medium
3    heroin    no       low
4    heroin    no       low
```

7.4.2 *Dropping variables*

Since we already have educ, there is no reason to keep our new variable workforce.years. Let's drop it. Notice the clever use of the minus sign.

```
CPS1 <- select(CPS85, -workforce.years)
head(CPS1, 1)
```

```
  wage educ race sex hispanic south married exper union age
1    9   10    W   M       NH    NS Married    27   Not  43
  sector
1  const
```

Any number of variables can be dropped or kept in this manner by supplying a vector of variables names.

```
CPS1 <- select(CPS85, c(workforce.years,exper))
```

Digging Deeper
Master programers in R such as Hadley Wickham, the author of the dplyr package, take advantage of special features of the language that allow such notation as minus to mean "exclude."

Columns can be specified by number as well as name (but this can be dangerous if you are wrong about where the columns are):

```
CPSsmall <- select(CPS85, select=1:4)
head(CPSsmall,2)
```

```
  select1 select2 select3 select4
1     9.0      10       W       M
2     5.5      12       W       M
```

The functions `matches()`, `contains()`, `starts_with()`, `ends_with()`, and `number_range()` are special functions that only work in the context of `select()` but can be useful for describing sets of variables to keep or discard.

```
head(select(HELPrct, contains("risk")), 2)
```

```
  drugrisk sexrisk
1        0       4
2        0       7
```

The nested functions in the previous command make the code a bit hard to read, and things would be worse if we were composing several more functions. The `magrittr` package (which loads when `dplyr` is loaded, hence when `mosaic` is loaded) provides an alternative syntax:

```
HELPrct %>% select(contains("risk")) %>% head(2)
```

```
  drugrisk sexrisk
1        0       4
2        0       7
```

The `%>%` operator uses the output from the left-hand side as the first input to the function on the right-hand side. This makes it easy to chain several data manipulation commands together in the order in which they are applied to the data without having to carefully nest parentheses and explicitly pass along outputs of one function as an argument to the next.

Here are a few more examples:

```
HELPrct %>% select(ends_with("e"))   %>% head(2)
```

```
  age female substance
1  37      0   cocaine
2  37      0   alcohol
```

```
HELPrct %>% select(starts_with("h")) %>% head(2)
```

```
  homeless
1   housed
2 homeless
```

```
HELPrct %>% select(matches("i[12]")) %>% head(2)  # regex matching
```

```
  i1 i2
1 13 26
2 56 62
```

7.4.3 *Renaming variables*

Both the column (variable) names and the row names of a data frames can be changed by simple assignment using `names()` or `row.names()`.

```
ddd           # small data frame we defined earlier
```

```
  number letter
1      1      a
2      2      b
3      3      c
4      4      d
5      5      e
```

```
# changing the row.names affects how a data.frame prints
row.names(ddd) <- c("Abe","Betty","Claire","Don","Ethel")
ddd
```

```
       number letter
Abe         1      a
Betty       2      b
Claire      3      c
Don         4      d
Ethel       5      e
```

It is also possible to reset just individual names with the following syntax.

```
# misspelled a name, let's fix it
row.names(ddd)[2] <- "Bette"
row.names(ddd)
```

```
[1] "Abe"    "Bette"  "Claire" "Don"    "Ethel"
```

The faithful data set (in the datasets package, which is always available) has very unfortunate names.

```
names(faithful)
```

```
[1] "eruptions" "waiting"
```

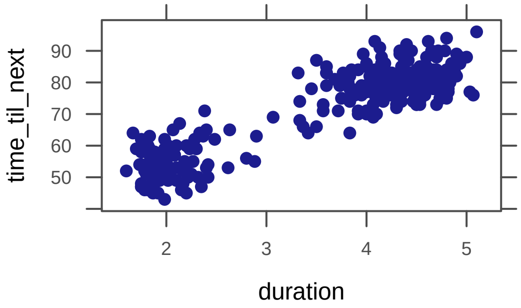
The measurements are the duration of an eruption and the time until the subsequent eruption, so let's give it some better names.

```
names(faithful) <- c('duration', 'time_til_next')
head(faithful, 3)
```

```
  duration time_til_next
1     3.60            79
2     1.80            54
3     3.33            74
```

Teaching Tip
An alternative solution is to use the geyser data set in the MASS package. The gyser data frame has better names and more data. But here we want to illustrate how to repair the damage in faithful.

```
xyplot(time_til_next ~ duration, faithful)
```



We can also rename a single variable using names(). For example, perhaps we want to rename educ (the second column) to education.

If the variable containing a data frame is modified or used to store a different object, the original data from the package can be recovered using data().

```
names(CPS85)[2] <- 'education'
CPS85[1,1:4]
```

```
  wage education race sex
1    9        10    W   M
```

If we don't know the column number (or generally to make our code clearer), a few more keystrokes produces

```
names(CPS85)[names(CPS85) == 'education'] <- 'educ'
CPS85[1,1:4]
```

```
  wage educ race sex
1    9   10    W   M
```

The `select()` function can also be used to rename variables.

See Section 7.4.2 for information that will make it clearer what is going on here.

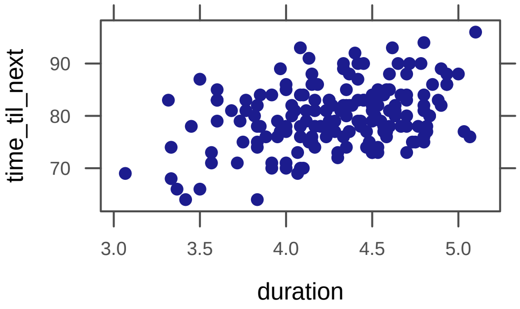
```
data(faithful)    # restore the original version
faithful2 <- faithful %>%
  select(duration=eruptions, time_til_next = waiting)
head(faithful2, 2)
```

```
  duration time_til_next
1      3.6            79
2      1.8            54
```

7.4.4 *Creating subsets*

We can use `filter()` to select only certain rows from a data frame.

```
# any logical can be used to create subsets
faithful2 %>% filter(duration > 3) -> faithfulLong
xyplot(time_til_next ~ duration, faithfulLong)
```



If all we want to do is produce a graph and don't need to save the subset, the plot above could also be made with one of the following

```
xyplot(time_til_next ~ duration,
       data = faithful2 %>% filter(duration > 3))
xyplot(time_til_next ~ duration, data = faithful2,
       subset=duration > 3)
```

7.4.5 *Summarising a data frame*

The summarise() (or summarize()) function summarizes a data frame as a single row.

```
HELPrct %>% summarise(x.bar = mean(age), s=sd(age))
```

```
  x.bar    s
1  35.7 7.71
```

This is especially useful in combination with group_by(), which divides the data frame into subsets. The following command will compute the mean and standard deviation for each subgroup defined by a different combination of sex and substance.

```
HELPrct %>% group_by(sex, substance) %>%
  summarise(x.bar = mean(age), s=sd(age))
```

```
Source: local data frame [6 x 4]
Groups: sex [?]

     sex substance x.bar     s
  (fctr)    (fctr) (dbl) (dbl)
1 female   alcohol  39.2  7.98
2 female   cocaine  34.9  6.20
3 female    heroin  34.7  8.04
4   male   alcohol  38.0  7.58
5   male   cocaine  34.4  6.89
6   male    heroin  33.1  7.97
```

The formula-based numerical summary functions supplied by the mosaic package are probably easier for this particular task, but using dplyr is more general.

```
favstats(age ~ sex + substance, data=HELPrct, .format="table")
```

```
   sex.substance min Q1 median Q3 max mean   sd   n missing
1 female.alcohol  23 33   37.0 45  58 39.2 7.98  36       0
2   male.alcohol  20 32   38.0 42  58 38.0 7.58 141       0
3 female.cocaine  24 31   34.0 38  49 34.9 6.20  41       0
4   male.cocaine  23 30   33.0 37  60 34.4 6.89 111       0
5  female.heroin  21 29   34.0 39  55 34.7 8.04  30       0
6    male.heroin  19 27   32.5 39  53 33.1 7.97  94       0
```

```
mean(age ~ sex + substance, data=HELPrct, .format="table")
```

```
           group mean
1 female.alcohol 39.2
2   male.alcohol 38.0
3 female.cocaine 34.9
4   male.cocaine 34.4
5  female.heroin 34.7
6    male.heroin 33.1
```

```
sd(age ~ sex + substance, data=HELPrct, .format="table")
```

```
           group   sd
1 female.alcohol 7.98
2   male.alcohol 7.58
3 female.cocaine 6.20
4   male.cocaine 6.89
5  female.heroin 8.04
6    male.heroin 7.97
```

7.4.6 *Arranging a data frame*

Sometimes it is convenient to reorder a data frame. We can do this with the arrange() function by specifying the variable(s) on which to do the sorting.

```
HELPrct %>%
  group_by(sex, substance) %>%
  summarise(x.bar = mean(age), s=sd(age)) %>%
  arrange(x.bar)
```

```
Source: local data frame [6 x 4]
Groups: sex [2]

     sex substance x.bar     s
  (fctr)    (fctr) (dbl) (dbl)
1 female    heroin  34.7  8.04
2 female   cocaine  34.9  6.20
3 female   alcohol  39.2  7.98
4   male    heroin  33.1  7.97
5   male   cocaine  34.4  6.89
6   male   alcohol  38.0  7.58
```

7.4.7 *Merging datasets*

The fusion1 data frame in the fastR package contains genotype information for a SNP (single nucleotide polymorphism) in the gene *TCF7L2*. The pheno data frame contains phenotypes (including type 2 diabetes case/control status) for an intersecting set of individuals. We can merge these together to explore the association between genotypes and phenotypes using one of the join functions in dplyr or using the merge() function.

```
require(fastR)
fusion1 %>% head(3)
```

```
     id     marker markerID allele1 allele2 genotype Adose Cdose Gdose Tdose
1  9735 RS12255372        1       3       3       GG     0     0     2     0
2 10158 RS12255372        1       3       3       GG     0     0     2     0
3  9380 RS12255372        1       3       4       GT     0     0     1     1
```

```
pheno %>% head(3)
```

```
    id     t2d  bmi sex  age smoker chol waist weight height   whr sbp dbp
1 1002    case 32.9   F 70.8 former 4.57 112.0   85.6    161 0.987 135  77
2 1009    case 27.4   F 53.9  never 7.32  93.5   77.4    168 0.940 158  88
3 1012 control 30.5   M 53.9 former 5.02 104.0   94.6    176 0.933 143  89
```

```
# merge fusion1 and pheno keeping only id's that are in both
fusion1m <- merge(fusion1, pheno, by.x='id', by.y='id',
                  all.x=FALSE, all.y=FALSE)
fusion1m %>% head(3)
```

```
    id     marker markerID allele1 allele2 genotype Adose Cdose Gdose Tdose     t2d  bmi
1 1002 RS12255372        1       3       3       GG     0     0     2     0    case 32.9
2 1009 RS12255372        1       3       3       GG     0     0     2     0    case 27.4
3 1012 RS12255372        1       3       3       GG     0     0     2     0 control 30.5
  sex  age smoker chol waist weight height   whr sbp dbp
1   F 70.8 former 4.57 112.0   85.6    161 0.987 135  77
2   F 53.9  never 7.32  93.5   77.4    168 0.940 158  88
3   M 53.9 former 5.02 104.0   94.6    176 0.933 143  89
```

```
pheno %>% left_join(fusion1, by="id") %>% dim()
```

```
[1] 2333   22
```

```
pheno %>% inner_join(fusion1, by="id") %>% dim()
```

```
[1] 2331   22
```

```
# which ids are only in \dataframe{pheno}?
setdiff(pheno$id, fusion1$id)
```

```
[1] 4011 9131
```

```
pheno %>% anti_join(fusion1, by="id")
```

```
    id     t2d  bmi sex age smoker chol waist weight height
1 4011    case 34.0   F  64  never 5.36   108   85.0    158
2 9131 control 26.7   M  73   <NA> 5.76    98   77.4    170
    whr sbp dbp
1 0.861 160  82
2 0.940 119  72
```

The difference between an inner join and a left join is that the inner join only includes rows from the first data frame that have a match in the second but a left join includes all rows of the first data frame, even if they do not have a match in the second. In the example above, there are two subjects in pheno that do not appear in fusion1.

merge() handles these distinctions with the all.x and all.y arguments. In this case, since the values are the same for each data frame, we could collapse by.x and by.y to by and collapse all.x and all.y to all. The first of these specifies which column(s) to use to identify matching cases. The second indicates whether cases in one data frame that do not appear in the other should be kept (TRUE) or dropped (filling in NA as needed) or dropped from the merged data frame.

Now we are ready to begin our analysis.

```
tally( ~ t2d + genotype + marker, data=fusion1m)

, , marker = RS12255372

         genotype
t2d        GG  GT  TT
  case    737 375  48
  control 835 309  27
```

7.5 Getting data from mySQL data bases

The RMySQL package allows direct access to data in MySQL data bases and the dplyr package facilitates processing this data in the same way as for data in a data frame. This makes it easy to work with very large data sets stored in public databases. The example below queries the UCSC genome browser to find all the known genes on chromosome 1.

UCSC — Univ. of California, Santa Cruz

```
# connect to a UCSC database
library(RMySQL)
UCSCdata <- src_mysql(
  host="genome-mysql.cse.ucsc.edu",
  user="genome",
  dbname="mm9")
# grab one of the many tables in the database
KnownGene <- tbl(UCSCdata, "knownGene")

# Get the gene name, chromosome, start and end sites for genes on Chromosome 1
Chrom1 <-
  KnownGene %>%
  select(name, chrom, txStart, txEnd) %>%
  filter(chrom == "chr1")
```

The resulting `Chrom1` is not a data frame, but behaves much like one.

```
class(Chrom1)
```

```
[1] "tbl_mysql" "tbl_sql"   "tbl"
```

```
Chrom1 %>%
  mutate(length=(txEnd - txStart)/1000) -> Chrom1l
Chrom1l
```

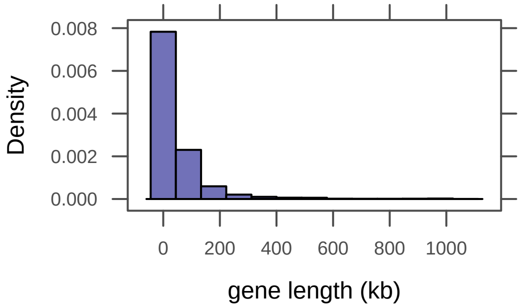
```
Source: mysql 5.6.26-log [genome@genome-mysql.cse.ucsc.edu:/mm9]
From: knownGene [3,056 x 5]
Filter: chrom == "chr1"

         name chrom txStart   txEnd length
        (chr) (chr)   (dbl)   (dbl)  (dbl)
1  uc007aet.1  chr1 3195984 3205713   9.73
2  uc007aeu.1  chr1 3204562 3661579 457.02
3  uc007aev.1  chr1 3638391 3648985  10.59
4  uc007aew.1  chr1 4280926 4399322 118.40
5  uc007aex.2  chr1 4333587 4350395  16.81
6  uc007aey.1  chr1 4481008 4483816   2.81
7  uc007aez.1  chr1 4481008 4486494   5.49
8  uc007afa.1  chr1 4481008 4486494   5.49
9  uc007afb.1  chr1 4481008 4486494   5.49
10 uc007afc.1  chr1 4481008 4486494   5.49
..        ...   ...     ...     ...    ...
```

CAUTION!
The arithmetic operations in this `mutate()` command are being executed in SQL, not in R, and the palette of allowable functions is much smaller. It is not possible, for example, to compute the logarithm of the length here using `log()`. For that we must first collect the data into a real data frame.

For efficiency, the full data are not pulled from the database until needed (or until we request this using `collect()`). This allows us, for example, to inspect the first few rows of a potentially large pull from the database without actually having done all of the work required to pull that data.

But certain things do not work unless we collect the results from the data based into an actual data frame. To plot the data using `lattice` or `ggplot2`, for example, we must first `collect()` it into a data frame.

```
Chrom1df <- collect(Chrom1l)       # collect into a data frame
histogram( ~ length, data=Chrom1df, xlab="gene length (kb)")
```



7.6 *Reshaping data with* `tidyr`

Sometimes data come in a shape that doesn't suit our purposes. The `tidyr` package includes several functions for tidying data, including `spread()` and `gather()`, which can be used to convert between "long" and "wide" formats . We may want to do this becuase of a change in perspective about what a unit of observation is, for example. For example, in the `traffic` data frame, each row is a year, and data for multiple states are provided.

```
traffic
```

```
  year cn.deaths   ny   cn   ma   ri
1 1951       265 13.9 13.0 10.2  8.0
2 1952       230 13.8 10.8 10.0  8.5
```

```
3 1953       275 14.4 12.8 11.0  8.5
4 1954       240 13.0 10.8 10.5  7.5
5 1955       325 13.5 14.0 11.8 10.0
6 1956       280 13.4 12.1 11.0  8.2
7 1957       273 13.3 11.9 10.2  9.4
8 1958       248 13.0 10.1 11.8  8.6
9 1959       245 12.9 10.0 11.0  9.0
```

We can reformat this so that each row contains a measurement for a single state in one year by gathering the states columns.

```
require(tidyr)
```

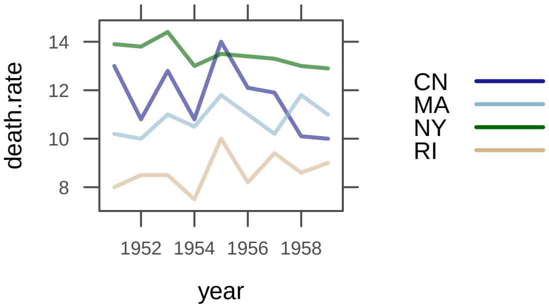
```
Loading required package:  tidyr
```

```
LongTraffic <-
  traffic %>%
  select(-cn.deaths) %>%
  gather(state, death.rate, ny:ri)
head(LongTraffic)
```

```
  year state death.rate
1 1951    ny       13.9
2 1952    ny       13.8
3 1953    ny       14.4
4 1954    ny       13.0
5 1955    ny       13.5
6 1956    ny       13.4
```

This long format allows us to create a plot like this.

```
xyplot(death.rate ~ year, data = LongTraffic, groups = toupper(state),
       type = "l",
       auto.key = list(space = "right", lines = TRUE, points = FALSE))
```



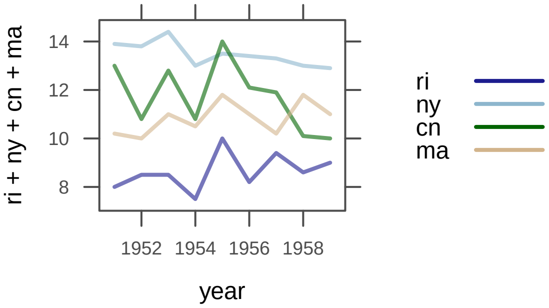
We can also reformat the other way, this time having all data for a given state form a row in the data frame.

```
StateTraffic <-
  LongTraffic %>%
  spread(state, death.rate)
StateTraffic %>% head(3)
```

```
  year   ny   cn   ma  ri
1 1951 13.9 13.0 10.2 8.0
2 1952 13.8 10.8 10.0 8.5
3 1953 14.4 12.8 11.0 8.5
```

We can create a plot using data in this format as well, but it involves a type of formula we have not seen before:

```
xplot(ri + ny + cn + ma ~ year, data=StateTraffic, type = "l",
       auto.key = list(space = "right", lines = TRUE, points = FALSE))
```



7.7 *Functions in* R

Functions in R have several components:

- a **name** (like `histogram`)[1]
- an ordered list of named **arguments** that serve as inputs to the function

  These are matched first by name and then by order to the values supplied by the call to the function. This is why we don't always include the argument name in our function calls. On the other hand, the availability of names means that we don't have to remember the order in which arguments are listed.

  Arguments often have **default values** which are used if no value is supplied in the function call.
- a **return value**

  This is the output of the function. It can be assigned to a variable using the assignment operator (`=`, `<-`, or `->`).
- **side effects**

  A function may do other things (like make a graph or set some preferences) that are not necessarily part of the return value.

When you read the help pages for an R function, you will see that they are organized in sections related to these components. The list of arguments appears in the **Usage** section along with any default values. Details about how the arguments are used appear in the **Arguments** section. The return value is listed in the **Value** section. Any side effects are typically mentioned in the **Details** section.

Now let's try writing our own function. Suppose you frequently wanted to compute the mean, median, and standard deviation of a distribution. You could make a function to do all three to save some typing.

Let's name our function `mystats()`. The `mystats()` will have one argument, which we are assuming will be a vector of numeric values. Here is how we could define it:

Even if you do not end up writing many functions yourself, writing a few functions will give you a much better feel for how information flows through R code.

```
mystats <- function(x) {
    mean(x)
    median(x)
    sd(x)
```

[1] Actually, it is possible to define functions without naming them; and for short functions that are only needed once, this can actually be useful.

```
}
```

```
mystats((1:20)^2)
```

```
[1] 128
```

There are ways to check the **class** of an argument to see if it is a data frame, a vector, numeric, etc. A really robust function should check to make sure that the values supplied to the arguments are of appropriate types.

The first line says that we are defining a function called `mystats()` with one argument, named `x`. The lines surrounded by curly braces give the code to be executed when the function is called. So our function computes the mean, then the median, then the standard deviation of its argument.

But as you see, this doesn't do exactly what we wanted. So what's going on? The value returned by the last line of a function is (by default) returned by the function to its calling environment, where it is (by default) printed to the screen so you can see it. In our case, we computed the mean, median, and standard deviation, but only the standard deviation is being returned by the function and hence displayed. So this function is just an inefficient version of `sd()`. That isn't really what we wanted.

We can use `print()` to print out things along the way if we like.

```
mystats <- function(x) {
    print(mean(x))
    print(median(x))
    print(sd(x))
}
```

```
mystats((1:20)^2)
```

```
[1] 144
[1] 110
[1] 128
```

Alternatively, we could use a combination of `cat()` and `paste()`, which would give us more control over how the output is displayed.

```
altmystats <- function(x) {
    cat(paste("  mean:", format(mean(x),4),"\n"))
    cat(paste(" edian:", format(median(x),4),"\n"))
    cat(paste("    sd:", format(sd(x),4),"\n"))
}
altmystats((1:20)^2)
```

```
  mean: 144
 edian: 110
    sd: 128
```

Either of these methods will allow us to see all three values, but if we try to store them ...

```
temp <- mystats((1:20)^2)
```

```
[1] 144
[1] 110
[1] 128
```

```
temp
```

```
[1] 128
```

A function in R can only have one return value, and by default it is the value of the last line in the function. In the preceding example we only get the standard deviation since that is the value we calculated last.

We would really like the function to return all three summary statistics. Our solution will be to store all three in a vector and return the vector.[2]

```
mystats <- function(x) {
        c(mean(x), median(x), sd(x))
}
mystats((1:20)^2)
```

```
[1] 144 110 128
```

Now the only problem is that we have to remember which number is which. We can fix this by giving names to the slots in our vector. While we're at it, let's add a few more favorites to the list. We'll also add an explicit `return()`.

[2] If the values had not all been of the same mode, we could have used a list instead.

```
mystats <- function(x) {
    result <- c(min(x), max(x), mean(x), median(x), sd(x))
    names(result) <- c("min","max","mean","median","sd")
    return(result)
}
mystats((1:20)^2)
```

```
   min    max   mean median     sd
     1    400    144    110    128
```

```
aggregate(Sepal.Length ~ Species, data=iris, FUN=mystats)
```

```
     Species Sepal.Length.min Sepal.Length.max
1     setosa            4.300            5.800
2 versicolor            4.900            7.000
3  virginica            4.900            7.900
  Sepal.Length.mean Sepal.Length.median Sepal.Length.sd
1             5.006               5.000           0.352
2             5.936               5.900           0.516
3             6.588               6.500           0.636
```

Notice how nicely this works with aggregate(). The favstats() function in the mosaic package includes the quartiles, mean, standard, deviation, sample size and number of missing observations.

```
favstats(Sepal.Length ~ Species, data=iris)
```

```
     Species min   Q1 median  Q3 max mean    sd  n missing
1     setosa 4.3 4.80    5.0 5.2 5.8 5.01 0.352 50       0
2 versicolor 4.9 5.60    5.9 6.3 7.0 5.94 0.516 50       0
3  virginica 4.9 6.23    6.5 6.9 7.9 6.59 0.636 50       0
```

We can get a version of our new function that works with the formula template like this.

```
# first create a version that works on vectors
mystats_ <- function(x, na.rm = TRUE) {
    result <- c(min(x, na.rm = na.rm), max(x, na.rm = na.rm), mean(x, na.rm = na.rm),
                median(x, na.rm = na.rm), sd(x, na.rm = na.rm))
    names(result) <- c("min","max","mean","median","sd")
    return(result)
}
# no create a version that knows the formula template
mystats <- aggregatingFunction1(mystats_, output.multiple = TRUE)
```

```
mystats(Sepal.Length ~ Species, data = iris)

     Species min max mean median    sd
1     setosa 4.3 5.8 5.01    5.0 0.352
2 versicolor 4.9 7.0 5.94    5.9 0.516
3  virginica 4.9 7.9 6.59    6.5 0.636
```

7.8 Sharing With and Among Your Students

Instructors often have their own data sets to illustrate points of statistical interest or to make a particular connection with a class. Sometimes you may want your class as a whole to construct a data set, perhaps by filling in a survey or by contributing their own small bit of data to a class collection. Students may be working on projects in small groups; it's nice to have tools to support such work so that all members of the group have access to the data and can contribute to a written report.

There are now many technologies that support such sharing. For the sake of simplicity, we will emphasize three that we have found particularly useful both in teaching statistics and in our professional collaborative work. These are:

- A web site with minimal overhead, such as provided by Dropbox.
- The services of Google Docs.
- A web-based RStudio server for R.

The first two are already widely used in university environments and are readily accessible simply by setting up accounts. Setting up an RStudio web server requires some IT support, but is well within the range of skills found in IT offices and even among some individual faculty.

TEACHING TIP
When accounts are set up on the RStudio server for a new class at Calvin, each user is given a symbolic link to a directory where the instructor can write files and students can only read files. This provides an easy way to make data, R code, or history files available to students from inside RStudio.

7.8.1 Using RStudio *server to share files*

The RStudio server runs on a Linux machine. Users of RStudio have accounts on the underlying Linux file system and it is possible to set up shared directories with permissions that allow multiple users to read and/or write files stored there. This has to be done outside of RStudio, but if you are familiar with the Linux operating system or have a system administrator who is willing to help you out, this is not difficult to do.

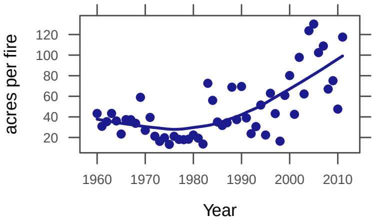
7.8.2 Your own web site

You may already have a web site. We have in mind a place where you can place files and have them accessed directly from the Internet. For sharing data, it's best if this site is public, that is, it does not require a login for others to access the files you put there. In this case, `read.file()` can read the data into R directly from the URL:

```
Fires <- read.csv("http://www.calvin.edu/~rpruim/data/Fires.csv")
head(Fires)
```

```
  Year Fires   Acres
1 2011 74126 8711367
2 2010 71971 3422724
3 2009 78792 5921786
4 2008 78979 5292468
5 2007 85705 9328045
6 2006 96385 9873745
```

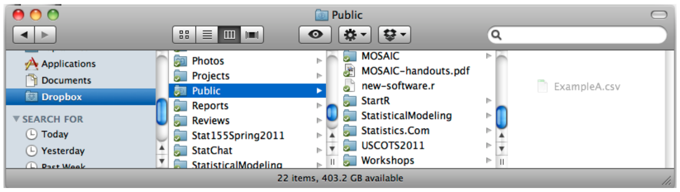
```
xplot(Acres/Fires ~ Year, data=Fires, ylab="acres per fire",
        type=c("p","smooth"))
```



Unfortunately, most "course support" systems such as Moodle orBlackboard do not provide such easy access to data. The Dropbox service for storing files in the "cloud" provides a very convenient way to distribute files over the web. (Go to `dropbox.com` for information and to sign up for a free account.) Dropbox is routinely used to provide automated backup and coordinated file access on multiple computers. But the Dropbox service also provides a `Public` directory. Any files that you place in that directory can be accessed directly by a URL.

To illustrate, suppose you wish to share some data set with your students. You've constructed this data set in a spreadsheet and stored it as a csv file, let's call it `example-A.csv`. Move this file into the `Public` directory under Dropbox — on most computers Dropbox arranges things so that its directories appear exactly like ordinary directories and you'll use the ordinary, familiar file management techniques such as drag and drop.

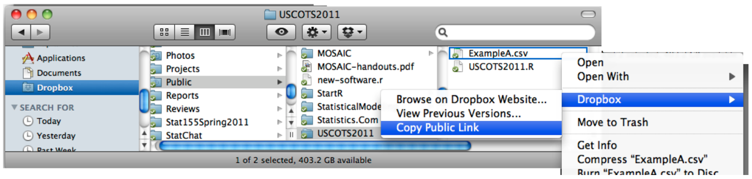
Our discussion of Dropbox is primarily for those who do not already know how to do this other ways.



Dragging a csv file to a Dropbox Public directory

Dropbox also makes it straightforward to construct the web-location identifying URL for any file by using mouse-based menu commands to place the URL into the clipboard, whence it can be copied to your course-support software system or any other place for distribution to students. For a csv file, reading the contents of the file into R can be done with the `read.csv()` function, by giving it the quoted URL:

```
a <- read.file("http://dl.dropbox.com/u/5098197/USCOTS2011/ExampleA.csv")
```



Getting the URL of a file in a Dropbox Public directory

This technique makes it easy to distribute data with little advance preparation. It's fast enough to do in the middle of a class: the csv file is available to your students (after a brief lag while Dropbox synchronizes). It can even be edited by you (but not by your students).

The same technique can be applied to all sorts of files like R workspaces or R scripts (files containing code). Of course, your students need to use the appropriate R command: load() for a workspace or source() for a script.

The example below will source a file that will print a welcoming message for you.

```
source('http://mosaic-web.org/go/R/hello.R')
```

```
Hello there.  You just sourced a file over the web!
```

But you can put any R code you like in the files you have your students source. You can install and load packages, retrieve or modify data sets, define new functions, or anything else R allows.

Many instructors will find it useful to create a file with your course-specific R scripts, adding on to it and modifying it as the course progresses. This allows you to distribute all sorts of special-purpose functions, letting you distribute new R material to your students. That brilliant new idea you had at 2 AM can be programmed up and put in place for your students to use the next morning in class. Then as you identify bugs and refine the program, you can make the updated software immediately available to your students.

If privacy is a concern, for instance if you want the data available only to your students, you can effectively accomplish this by giving files names known only to your students, e.g., Example-A78r423.csv.

Caution!
Security through Obscurity of this sort will not generally satisfy institutional data protection regulations nor professional ethical requirements, so nothing truly sensitive or confidential should be “protected" in this manner.

7.8.3 GoogleDocs

The Dropbox technique (or any other system of posting files to the Internet) is excellent for broadcasting: taking files you create and distributing them in a read-only fashion to your students. But when you want two-way or multi-way sharing of files, other techniques are called for, such as provided by the GoogleDocs service.

GoogleDocs allows students and instructors to create various forms of documents, including reports, presentations, and spreadsheets. (In addition to creating documents *de novo*, Google will also convert existing documents in a variety of formats.)

Once on the GoogleDocs system, the documents can be edited *simultaneously* by multiple users in different locations. They can be shared with individuals or groups and published for unrestricted viewing and even editing.

For teaching, this has a variety of uses:

- Students working on group projects can all simultaneously have access to the report as it is being written and to data that is being assembled by the group.
- The entire class can be given access to a data set, both for reading and for writing.
- The Google Forms system can be used to construct surveys, the responses to which can populate a spreadsheet that can be read back into RStudio by the survey creators.
- Students can "hand in" reports and data sets by copying a link into a course support system such as Moodle or Blackboard, or emailing the link.
- The instructor can insert comments and/or corrections directly into the document.

An effective technique for organizing student work and ensuring that the instructor (and other graders) have access to it, is to create a separate Google directory for each student in your class (Dropbox can also be used in this manner). Set the permission on this directory to share it with the student. Anything she or he drops into the directory is automatically available to the instructor. The student can also share with specific other students (e.g., members of a project group).

Data can be read directly from google sheets using the `googlesheets` package. This works much like `read_excel()` from the `readxl` package.

7.9 Additional Notes on R Syntax

7.9.1 Text and Quotation Marks

For the most part, text in R must be enclosed in either single or double quotations. It usually doesn't matter which you use, unless you want one or the other type of quotation mark *inside* your text. Then you should use the other type of quotation mark to mark the beginning and the end.

```
# apostrophe inside requires double quotes around text
text1 <- "Mary didn't come"
# this time we flip things around
text2 <- 'Do you use "scare quotes"?'
```

7.10 Common Error Messages and What Causes Them

7.10.1 Error: Object not found

R reports that an object is not found when it cannot locate an object with the name you have used. One common reason for this is a typing error. This is easily corrected by retyping the name with the correct spelling.

```
histogram( ~ aeg, data=HELPrct)
```

```
Error in eval(expr, envir, enclos):  object 'aeg' not
found
```

Another reason for an object-not-found error is using unquoted text where quotation marks were required.

```
text3 <- hello
```

```
Error in eval(expr, envir, enclos):  object 'hello' not
found
```

In this case, R is looking for some object named `hello`, but we meant to store a string:

```
text3 <- "hello"
```

7.10.2 Error: unexpected . . .

If while R is parsing a statement it encounters something that does not make sense it reports that something is “unexpected”. Often this is the result of a typing error – like omitting a comma.

```
c(1,2 3)                        # missing a comma
```

```
Error in c():  unexpected numeric constant in "c(1,2 3"
```

7.10.3 Error: object of type ‘closure’ is not subsettable

The following produces an error if `time` has not been defined.

```
time[3]
```

```
Error in time[3]:  object of type ’closure’ is not
subsettable
```

There is a function called `time()` in R, so if you haven’t defined a vector by that name, R will try to subset the `time()` function, which doesn’t make sense.

Typically when you see this error, you have a function in a place you don’t mean to have a function. The message can be cryptic to new users because of the reference to a closure.

7.10.4 Other Errors

If you encounter other errors and cannot decipher them, often pasting the error message into a google search will find a discussion of that error in a context where it stumped someone else.

7.11 Review of R Commands

Here is a brief summary of the commands introduced in this chapter.

```
source("file.R")                                 # execute commands in a file

x <- 1:10                                        # create vector with numbers 1 through 10
M <- matrix(1:12, nrow=3)                        # create a 3 x 4 matrix
data.frame(number = 1:26, letter=letters[1:26])  # create a data frame


mode(x)                                  # returns mode of object x
length(x)                                # returns length of vector or list
dim(HELPrct)                             # dimension of a matrix, array, or data frame
nrow(HELPrct)                            # number of rows
ncol(HELPrct)                            # number of columns
names(HELPrct)                           # variable names in data frame
row.names(HELPrct)                       # row names in a data frame
attributes(x)                            # returns attributes of x


toupper(x)                               # capitalize
as.character(x)                          # convert to a character vector
as.logical(x)                            # convert to a logical (TRUE or FALSE)
as.numeric(x)                            # convert to numbers
as.integer(x)                            # convert to integers
factor(x)                                # convert to a factor [categorical data]
class(x)                                 # returns class of x


smallPrimes <- c(2,3,5,7,11)             # create a (numeric) vector
rep(1, 10)                               # ten 1's
seq(2, 10, by=2)                         # evens less than or equal to 10
rank(x)                                  # ranks of items in x
sort(x)                                  # returns elements of x in sorted order
order(x)                                 # x[ order(x) ] is x in sorted order
rev(x)                                   # returns elements of x in reverse order
diff(x)                                  # returns differences between consecutive elements
paste("Group", 1:3, sep="")              # same as c("Group1", "Group2", "Group3")
```

```
write.table(HELPrct, file="myHELP.txt")          # write data to a file
write.csv(HELPrct, file="myHELP.csv")            # write data to a csv file
save(HELPrct, file="myHELP.Rda")                 # save object(s) in R's native format

modData <- HELPrct %>% mutate(old = age > 50)    # add a new variable to data frame
women <- HELPrct %>% filter(sex=='female')       # select only specified cases
favs <- HELPrct %>% select(age, sex, substance)  # keep only 3 columns

trellis.par.set(theme=col.mosaic())             # choose theme for lattcie graphics
show.settings()                                 # inspect lattice theme
```

7.12 Exercises

7.1 Using `faithful` data frame, make a scatter plot of eruption duration times vs. the time since the previous eruption.

7.2 The `fusion2` data set in the `fastR` package contains genotypes for another SNP. Merge `fusion1`, `fusion2`, and `pheno` into a single data frame.

Note that `fusion1` and `fusion2` have the same columns.

```
names(fusion1)

 [1] "id"       "marker"   "markerID" "allele1"  "allele2" 
 [6] "genotype" "Adose"    "Cdose"    "Gdose"    "Tdose"  

names(fusion2)

 [1] "id"       "marker"   "markerID" "allele1"  "allele2" 
 [6] "genotype" "Adose"    "Cdose"    "Gdose"    "Tdose"  
```

You may want to use the `suffixes` argument to `merge()` or rename the variables after you are done merging to make the resulting data frame easier to navigate.

Tidy up your data frame by dropping any columns that are redundant or that you just don't want to have in your final data frame.

8
Getting Interactive: manipulate and shiny

One very attractive feature of RStudio is the manipulate() function (in the manipulate package, which is only available within RStudio). This function makes it easy to create a set of controls (such as sliders, checkboxes, drop down selections, etc.) that can be used to dynamically change values within an expression. When a value is changed using these controls, the expression is automatically re-executed and any plots created as a result are redrawn. This can be used to quickly prototype a number of activities and demos as part of a statistics lecture.

shiny is a new web development system for R being designed by the RStudio team. shiny uses a reactive programming model to make it relatively easy for an R programmer to create highly interactive, well designed web applications using R without needing to know much about web programming. Programming in shiny is more involved than using manipulate, but it offers the designer more flexibility. One of the goals in creating shiny was to support corporate environments, where a small number of statisticians and programmers can create web applications that can be used by others within the company without requiring them to know any R. This same framework offers many possibilities for educational purposes as well. Some have even suggested implementing fairly extensive GUI interfaces to commonly used R functionality using shiny.

8.1 Getting Started with `manipulate`

The `manipulate()` function and the various control functions that are used with it are only available after loading the `manipulate` package, which is only available in RStudio.

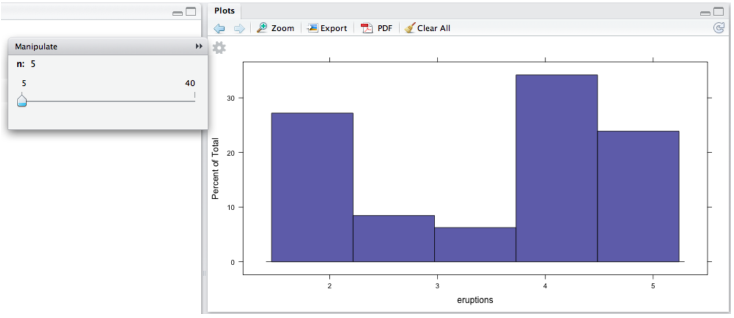
```
require(manipulate)
```

8.1.1 Sliders

```
manipulate(
  histogram( ~ eruptions, data=faithful, n=N),
  N = slider(5,40)
)
```

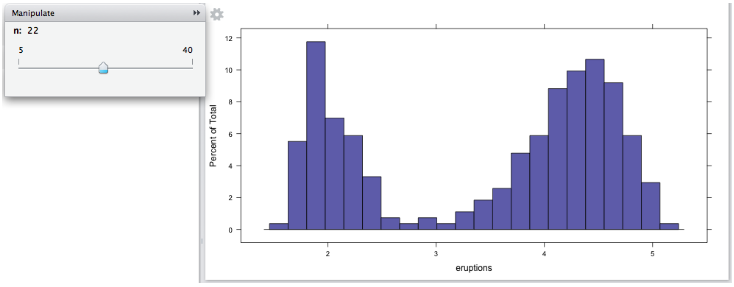
This generates a plot along with a slider ranging from 5 bins to 40.

We find it useful to capitalize the inputs to the manipulated expression that are hooked up to `manipulate` controls. This helps avoid naming collisions and signals how the main manipulated expression is being used.



n: 5

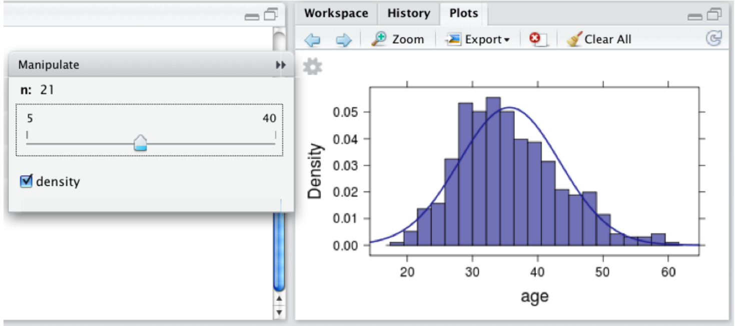
When the slider is changed, we see a clearer view of the eruptions of Old Faithful.



n: 22

8.1.2 Check Boxes

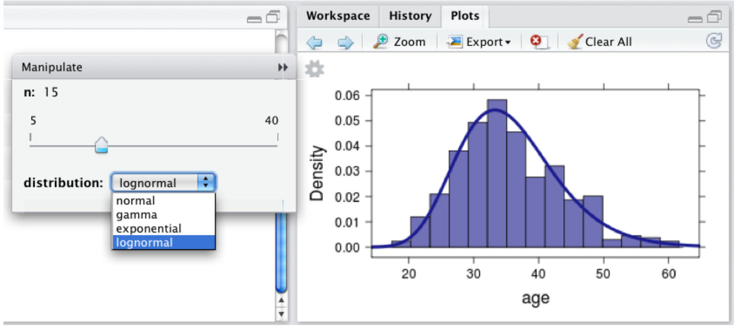
```
manipulate(
  histogram( ~ age, data=HELPrct, n=N, density=DENSITY),
  N = slider(5,40),
  DENSITY = checkbox()
)
```



8.1.3 Drop-down Menus

Drop-down menus can be added using the picker() function.

```
manipulate(
    histogram( ~ age, data=HELPrct, n=N,
                        fit=DISTRIBUTION, dlwd=4),
    N = slider(5,40),
    DISTRIBUTION =
        picker('normal', 'gamma', 'exponential', 'lognormal',
               label="distribution")
)
```

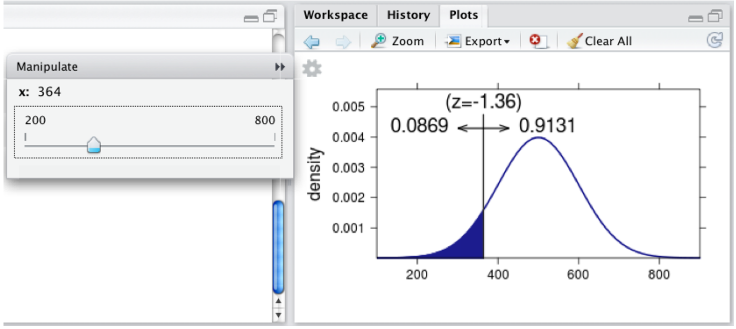


8.1.4 *Visualizing Normal Distributions*

In this section we will gradually build up a small `manipulate` example that shows the added flexibility that comes from writing a function that returns a `manipulate` object. Such functions can be distributed to students to allow them to explore interactively in a more flexible way.

We begin by creating an illustration of tail probabilities in a normal distribution.

```
manipulate(
    xpnorm( X, 500, 100, verbose=FALSE, invisible=TRUE ),
    X = slider(200,800) )
```



x: 364

The version below can be used to investigate central probabilities and tail probabilities.

```
manipulate(
    xpnorm( c(-X,X), 500, 100, verbose=FALSE, invisible=TRUE ),
    X = slider(200,800) )
```

These examples work with a fixed distribution. Here is a fancier version in which a function returns a manipulate object. This allows us to easily create illustrations like the ones above for any normal distribution.

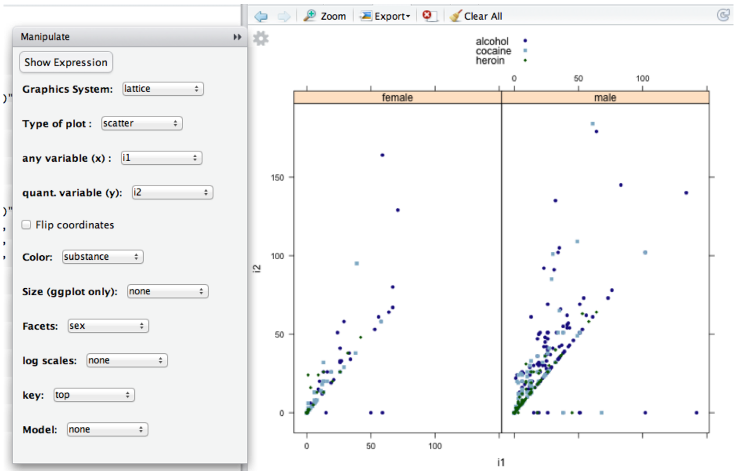
```
mNorm <- function( mean=0, sd=1 ) {
  lo <- mean - 5*sd
  hi <- mean + 5*sd
  manipulate(
    xpnorm( c(A,B), mean, sd, verbose=FALSE, invisible=TRUE ),
    A = slider(lo, hi, initial=mean-sd),
    B = slider(lo, hi, initial=mean+sd)
  )
}
mNorm( mean=100, sd=10 )
```

8.2 mPlot()

The mosaic package provides the mPlot() function which allows users to create a wide variety of plots using either lattice or ggplot2. Furthermore, the code used to generate these plots can be displayed upon request. This facilitates learning these commands, allows users to make further modifications that are not possible in the manipulate interface, and provides an easy copy-and-paste mechanism for dropping these plots into other documents.

The available plots come in two clusters, depending on whether the underlying plot is essentially two-variable or one-variable. Additional variables can be represented using color, size, and sub-plots (facets).

```
# These are essentially 2-variable plots
mPlot( HELPrct, "scatter" )          # start with a scatter plot
mPlot( HELPrct, "boxplot" )          # start with boxplots
mPlot( HELPrct, "violin" )           # start with violin plots
# These are essentially 1-variables plots
mPlot( HELPrct, "histogram" )        # start with a histogram
mPlot( HELPrct, "density" )          # start with a density plot
mPlot( HELPrct, "frequency polygon" )  # start with a frequency polygon
```



Graphics System: lattice

Type of plot : scatter

any variable (x) : i1

quant. variable (y): i2

☐ Flip coordinates

Color: substance

Size (ggplot only): none

Facets: sex

log scales: none

key: top

Model: none

8.3 Shiny

shiny is a package created by the RStudio team to, in their words,

> [make] it incredibly easy to build interactive web applications with R. Automatic “reactive" binding between inputs and outputs and extensive pre-built widgets make it possible to build beautiful, responsive, and powerful applications with minimal effort.

These web applications can, of course, run R code to do computations and produce graphics that appear in the web page.

The level of coding skill required to create this is beyond the scope of this book, but those with a little more programming background can easily learn the necessary toolkit to make beautiful interactive web pages. More information about shiny and some example applications are available at http://www.rstudio.com/shiny/.

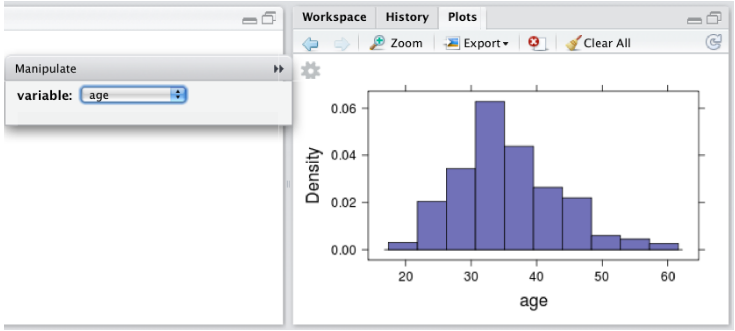
Exercises

8.1 The following code makes a scatterplot with separate symbols for each sex.

```
xyplot(cesd ~ age, data=HELPrct, groups=sex)
```

Build a manipulate example that allows you to turn the grouping on and off with a checkbox.

8.2 Build a manipulate example that uses a picker to select from a number of variables to make a plot for. Here’s an example with a histogram:



Manipulate

variable: age

8.3 Design your own interactive demonstration idea and implement it using RStudio `manipulate` tools.

Bibliography

[CR15] Angelo Canty and Brian Ripley. *boot: Bootstrap R (S-Plus) Functions*, 2015. R package version 1.3-17.

[DH97] A. C. Davison and D. V. Hinkley. *Bootstrap Methods and Their Applications*. Cambridge University Press, 1997.

[Fis25] R. A. Fisher. *Statistical Methods for Research Workers*. Oliver & Boyd, 1925.

[Fis70] R. A. Fisher. *Statistical Methods for Research Workers*. Oliver & Boyd, 14th edition, 1970.

[Hes15a] Tim Hesterberg. *resample: Resampling Functions*, 2015. R package version 0.4.

[Hes15b] Tim C. Hesterberg. What teachers should know about the bootstrap: Resampling in the undergraduate statistics curriculum. *The American Statistician*, 2015.

[LLM12] Robin H Lock, Patti Frazer Lock, and Kari Lock Morgan. *Statistics: Unlocking the Power of Data*. Wiley Global Education, 2012.

[NT10] D. Nolan and D. Temple Lang. Computing in the statistics curriculum. *The American Statistician*, 64(2):97–107, 2010.

[Sal01] D. Salsburg. *The Lady Tasting Tea: How statistics revolutionized science in the twentieth century*. W.H. Freeman, New York, 2001.

[TCC$^+$15a] N. Tintle, B. Chance, G. Cobb, S. Roy, T. Swanson, and J. VanderStoep. Combating anti-statistical thinking using simulation-based methods throughout the undergraduate curriculum. *The American Statistician*, 69(4), 2015.

[TCC$^+$15b] Nathan Tintle, Beth Chance, George Cobb, Allan Rossman, Soma Roy, Todd Swanson, and Jill VanderStoep. *Introduction to Statistical Investigations*. Wiley Global Education, 2015.

[Wor14] ASA Undergraduate Guidelines Workgroup. 2014 curriculum guidelines for undergraduate programs in statistical science. Technical report, American Statistical Association, November 2014. `http://www.amstat.org/education/curriculumguidelines.cfm`.

Index